pyrolite Documentation

Morgan Williams

Nov 15, 2026

GETTING STARTED

1 Introduction 1

2 Publication 3

3 Why *pyrolite*? 5

4 Getting Started 7

5 Development 159

6 Reference 189

Bibliography 327

Python Module Index 329

Index 331

CHAPTER

ONE

INTRODUCTION

pyrolite is a set of tools for making the most of your geochemical data.

The python package includes functions to work with compositional data, to transform geochemical variables (e.g. elements to oxides), functions for common plotting tasks (e.g. spiderplots, ternary diagrams, bivariate and ternary density diagrams), and numerous auxiliary utilities.

- On this site you can browse the *API*, or look through some of the *usage examples*.
- There's also a quick *installation guide*, a list of *recent changes* and some notes on where the project is heading in the *roadmap*.
- If you're interested in *contributing to the project*, there are many potential avenues, whether you're experienced with python or not.

CHAPTER

TWO

PUBLICATION

An overview of an early version of pyrolite was published in the Journal of Open Source Software, and a publication focusing on using `lambdas` and tetrads to parameterise Rare Earth Element patterns has been published in Mathematical Geosciences.

See also

Citation

CHAPTER

THREE

WHY *PYROLITE*?

The name *pyrolite* is an opportunistic repurposing of a term used to describe an early model mantle composition proposed by Ringwood[1], comprised principally of **pyr**-oxene & **ol**-ivine. While the model certainly hasn't stood the test of time, the approach optimises the aphorism "All models are wrong, but some are useful"[2]. It is with this mindset that pyrolite is built, to better enable you to make use of your geochemical data to build and test geological models.

[1] Ringwood, A.E. (1962). A model for the upper mantle. Journal of Geophysical Research (1896-1977) 67, 857–867. doi: 10.1029/JZ067i002p00857

[2] Box, G.E.P. (1976). Science and Statistics. Journal of the American Statistical Association 71, 791–799. doi: 10.1080/01621459.1976.10480949

CHAPTER

FOUR

GETTING STARTED

4.1 Installation

pyrolite is available on PyPi, and can be downloaded with pip:

```
pip install --user pyrolite
```

Note

With pip, a user-level installation is generally recommended (i.e. using the `--user` flag), especially on systems which have a system-level installation of Python (including `*.nix`, WSL and MacOS systems); this can avoid some permissions issues and other conflicts. For detailed information on other pip options, see the relevant docs. pyrolite is not yet packaged for Anaconda, and as such `conda install pyrolite` will not work.

4.1.1 Upgrading pyrolite

New versions of pyrolite are released frequently. You can upgrade to the latest edition on PyPi using the `--upgrade` flag:

```
pip install --upgrade pyrolite
```

See also

Development Installation

4.1.2 Optional Dependencies

Optional dependencies (`skl`, `spatial`, `db`, `stats`, `notebook`, `excel`) can be specified during `pip` installation. For example:

```
pip install --user pyrolite[stats]

pip install --user pyrolite[skl,stats,notebook]
```

4.2 Getting Started

4.2.1 Getting Set Up

Before you can get up and running with pyrolite, you'll need to have a distribution of Python installed. If you're joining the scientific Python world, chances are that the Anaconda distributions are a good match for you[1]. Alternatively, the recommendation from our side would be to use uv. Check PyPI for the most up to date information regarding `pyrolite` compatibility, but as of writing this guide, `pyrolite` is known to work well with Python 3.10 and up.

When it comes to how you edit files and interact with Python, there are now many different choices (especially for editors), and choosing something which allows you to work the way you wish to work is the key aspect. While you can certainly edit python files (`.py`) in any text editor, choosing an editor designed for the task makes life easier. Choosing one is subjective - know that many exist, and perhaps try a few. Integrated Development Environments (IDEs) often allow you to quickly edit and run code within the same window (e.g. Spyder, which is typically included in the default Anaconda distribution), or the commonly used Visual Studio Code. Through notebooks and related ideas the Jupyter ecosystem has broadened how people are interacting with code across multiple languages, including Python; you can edit and run Jupyter notebooks in both VS Code and Python.

Finally, consider getting up to speed with simple Git practises for your projects and code such that you can keep versioned histories of your analyses, and have a look at hosted repository services (e.g. GitHub, GitLab, Codeberg). These hosted repositories together with integrated services are often worth taking advantage of (e.g. hosting material and analyses from papers, posters or presentation, and linking this through to Zenodo to get an archived version with a DOI).

4.2.2 Installing pyrolite

There's a separate page dedicated to *pyrolite installations*, but for most purposes, the best way to install pyrolite is through opening a terminal and type:

```
pip install pyrolite
```

To keep pyrolite up to date (new versions are released often), periodically run the following to update your local version:

```
pip install --upgrade pyrolite
```

4.2.3 Writing Some Code

If you're new to Python[2], or just new to `pyrolite`, checking out some of the examples is a good first step. You should be able to copy-paste or download and run each of these examples as written on your own system, or alternatively you can run them interatively in your browser thanks to Binder and sphinx-gallery (check for the links towards the bottom of each page). Have a play with these, and adapt them to your own purposes.

4.2.4 Importing Data

A large part of the pyrolite API is based around `pandas` DataFrames. One of the first hurdles for new users is importing their own data tables. To make this as simple as possible, it's best to organise - or 'tidy' - your data tables[3]. Minimise unnecessary whitespace, and where possible make sure your table columns are the first row of your table. In most

[1] If you're strapped for space, need to avoid licenscing restrictions, or are bloat-averse, you could also consider using Anaconda's miniconda distributions.

[2] If you're completely new to Python, check out some of the many free online courses to get up to scratch with basic Python concepts, data structures and get in a bit of practice writing code. Knowing your way around some of these things before you dive into applying them can help make it a much more surmountable challenge. Remember that the pyrolite community is also around to help out if you get stuck, and we all started from a similar place! There are no 'stupid questions', so feel free to ping us on Gitter with any questions or aspects that are proving particularly challenging.

[3] Where each variable is a column, and each observation is a row. If you're unfamiliar with the 'Tidy Data' concept, check out[4].

[4] Wickham, H., 2014. Tidy Data. Journal of Statistical Software 59, 1–23. doi: doi.org/10.18637/jss.v059.i10

cases, where these data are in the form of text or Excel files, the typical steps for data import are similar. A few simple examples are given below.

To import a table from a .csv file:

```
from pathlib import Path
import pandas as pd

filepath = Path('./mydata.csv')
df = pd.read_csv(filepath)
```

In the case of an excel table:

```
filepath = Path('./mydata.xlsx')
df = pd.read_excel(filepath)
```

There is also a pyrolite function which abstracts away these differences by making a few assumptions, and enables you to import the table from either a csv or excel file:

```
from pyrolite.util.pd import read_table
df = read_table(filepath)
```

4.2.5 Gitter Community

A Gitter Community has been set up to serve as a landing page for conversations, questions and support regarding the pyrolite python package and related activities. Here we hope to capture questions and bugs from the community such that these can be addressed quickly, and we can ensure pyrolite and its documentation are as useful as possible to the community. Please feel free to use this space to:

- Ask questions or seek help about getting started with pyrolite or particular pyrolite features
- Get tips for troubleshooting
- Discuss the general development of pyrolite
- Ask about contributing to pyrolite

Items which are related to specific aspects of pyrolite development (requesting a feature, or reporting an identified bug) are best coordinated through GitHub, but feel free to touch base here first. See below and the *contribution* and *development* guides for more information.

Note that users and contributors in online spaces (including Gitter) are expected to adhere to the *Code of Conduct* of this project (and any other guidelines of the relevant platform).

4.2.6 Bugs, Debugging & Bug Reporting

This section provides some guidance for what to do when things don't work as expected, and some tips for debugging issues and common errors associated with pyrolite. Note that the scope of these suggestions is necessarily limited, and specific issues which relate more to `matplotlib.pyplot`, `pandas`, and `numpy` are often well documented elsewhere online.

- Checked the documentation, had a look for FAQ and examples here and still stuck?

  The maintainers are happy to answer questions and help you solve small bugs. It's useful to know where people get stuck so we can modify things where useful, and this is an easy way to contribute to the project. Consider posting a question on Gitter, and if you think it's something others may run into or could be a problem related to use of another package, perhaps also consider posting a question on stackoverflow for visibility.

- Think it's a bug or problem with pyrolite specifically?

  Some guidelines for reporting issues and bugs are given in the *contributing guide*.

See also

Examples, *API*, *Changelog*, *Code of Conduct*

4.3 Examples

This example gallery includes a variety of examples for using pyrolite which you can copy, download and alter, or run on Binder.

4.4 Plotting Examples

pyrolite provides some functionality for basic plotting of geochemical data in the form of spidergrams (pyrolite.plot.spider), ternary diagrams (pyrolite.plot.tern) and density diagrams (i.e. 2D histograms, pyrolite.plot.density).

4.4.1 Stem Plots

Stem plots are commonly used to visualise discrete distributions of data, and are useful to highlight discrete observations where the precision of values along one axis is high (e.g. an independent spatial measure like depth) and the other is less so (such that the sampling frequency along this axis is important, which is not emphasised by a scatter plot).

```
import numpy as np
import pandas as pd

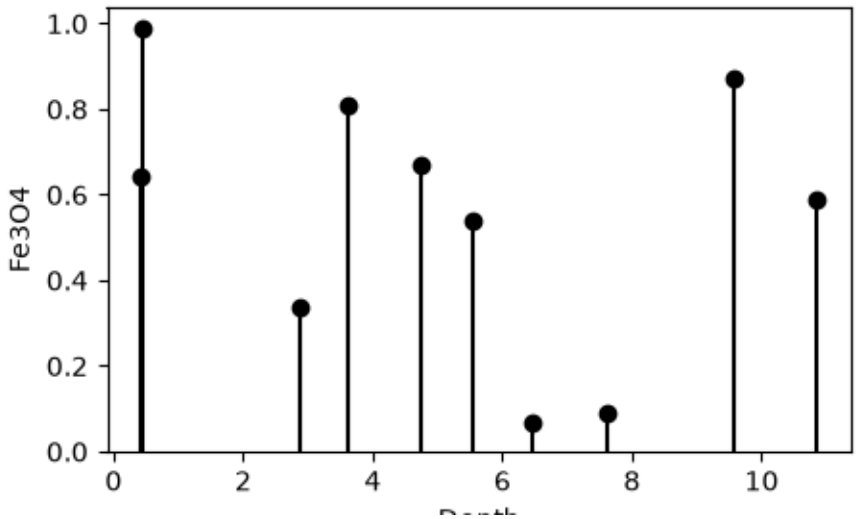

np.random.seed(82)
```

First let's create some example data:

```
x = np.linspace(0, 10, 10) + np.random.randn(10) / 2.0
y = np.random.rand(10)
df = pd.DataFrame(np.vstack([x, y]).T, columns=["Depth", "Fe3O4"])
```

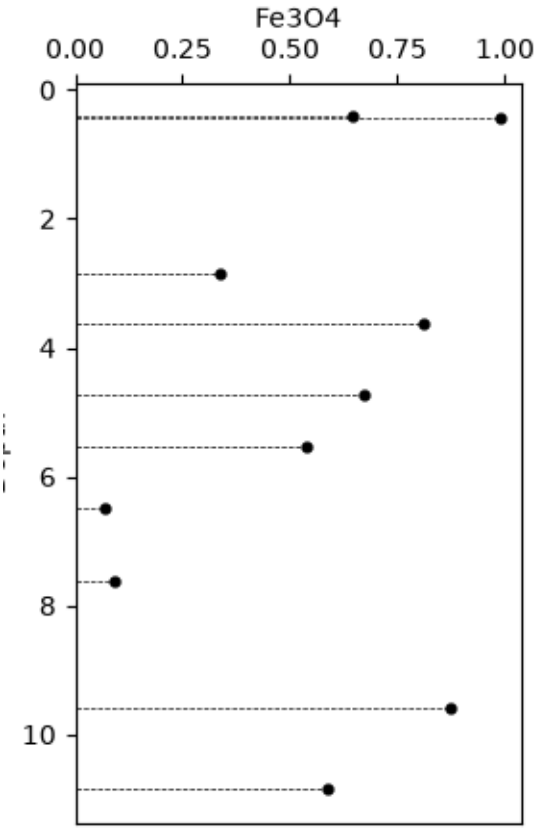
A minimal stem plot can be constructed as follows:

```
ax = df.pyroplot.stem(color="k", figsize=(5, 3))
```



Stem plots can also be used in a vertical orientation, such as for visualising discrete observations down a drill hole:

```
ax = df.pyroplot.stem(
    orientation="vertical",
    s=12,
    linestyle="--",
    linewidth=0.5,
    color="k",
    figsize=(3, 5),
)
# the yaxes can then be inverted using:
ax.invert_yaxis()
# and if you'd like the xaxis to be labeled at the top:
ax.xaxis.set_ticks_position("top")
ax.xaxis.set_label_position("top")
```



Total running time of the script: (0 minutes 0.105 seconds)

4.4.2 Ternary Plots

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

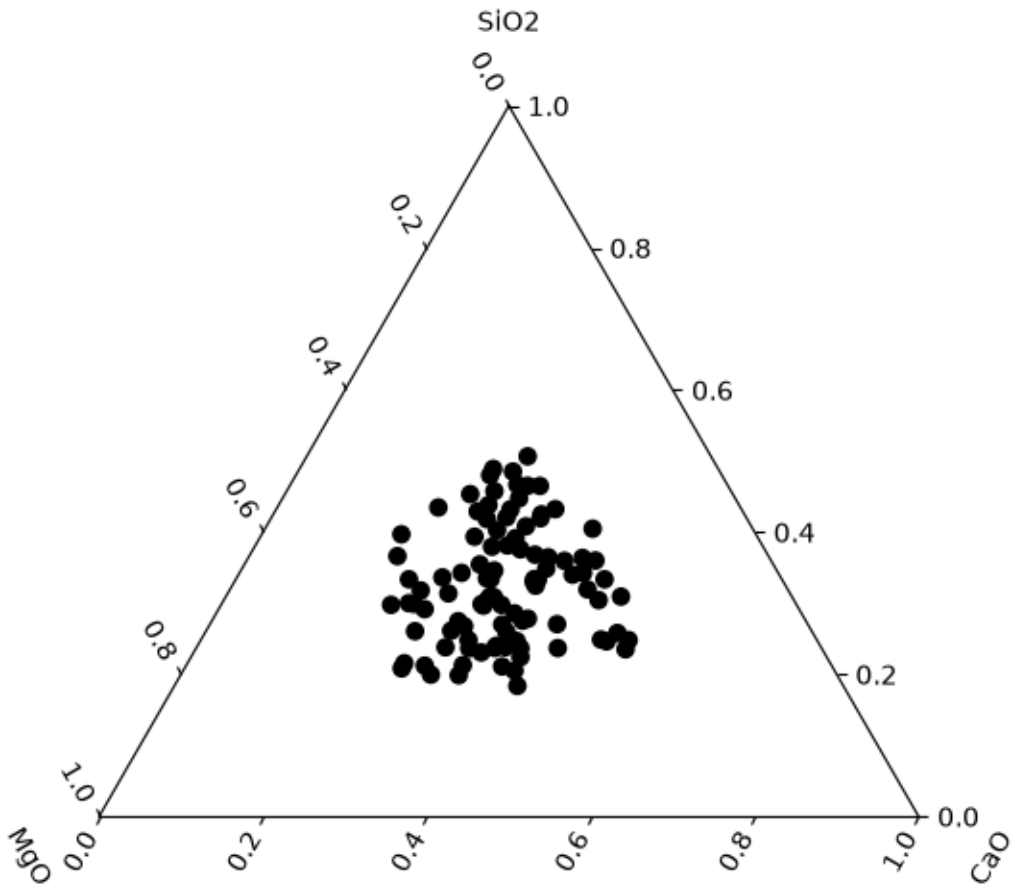
np.random.seed(82)
```

Let's first create some example data:

```
df = pd.DataFrame(data=np.exp(np.random.rand(100, 3)), columns=["SiO2", "MgO", "CaO"])
df.loc[:, ["SiO2", "MgO", "CaO"]].head()
```

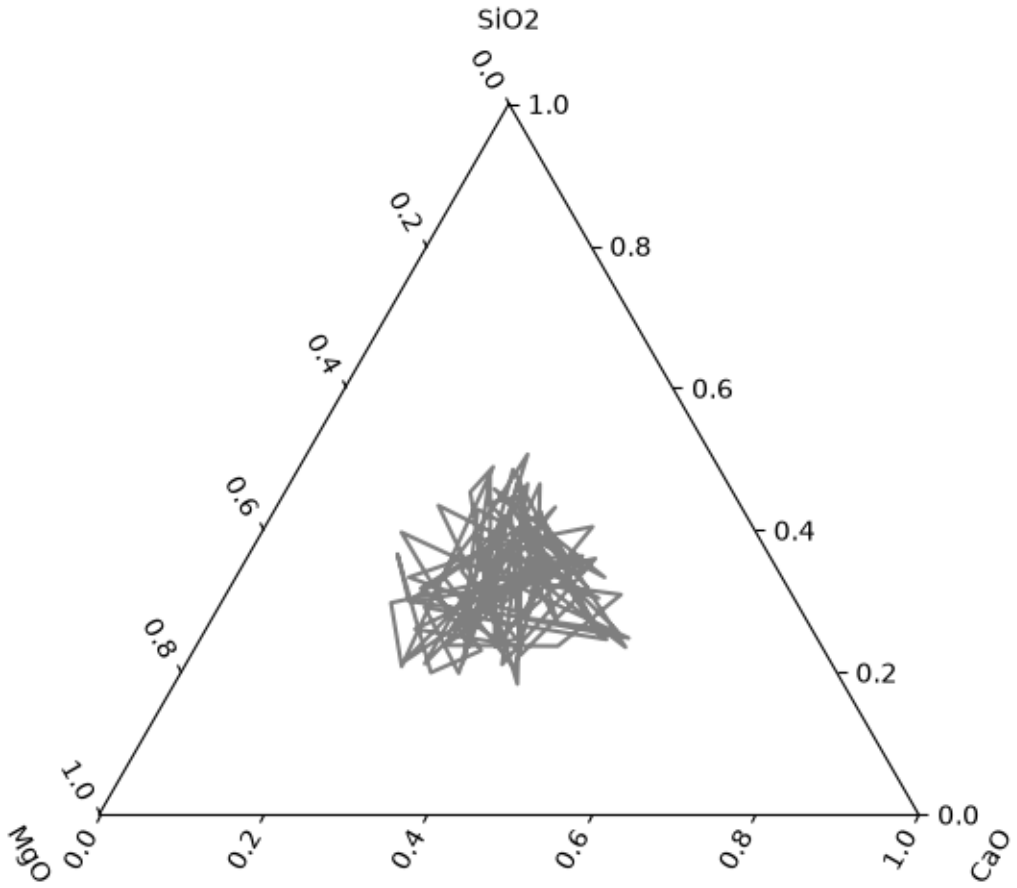
Now we can create a simple scatter plot:

```
ax = df.loc[:, ["SiO2", "MgO", "CaO"]].pyroplot.scatter(c="k")
plt.show()
```



If the data represent some continuous series, you could also plot them as lines:

```
ax = df.loc[:, ["SiO2", "MgO", "CaO"]].pyroplot.plot(color="k", alpha=0.5)
plt.show()
```

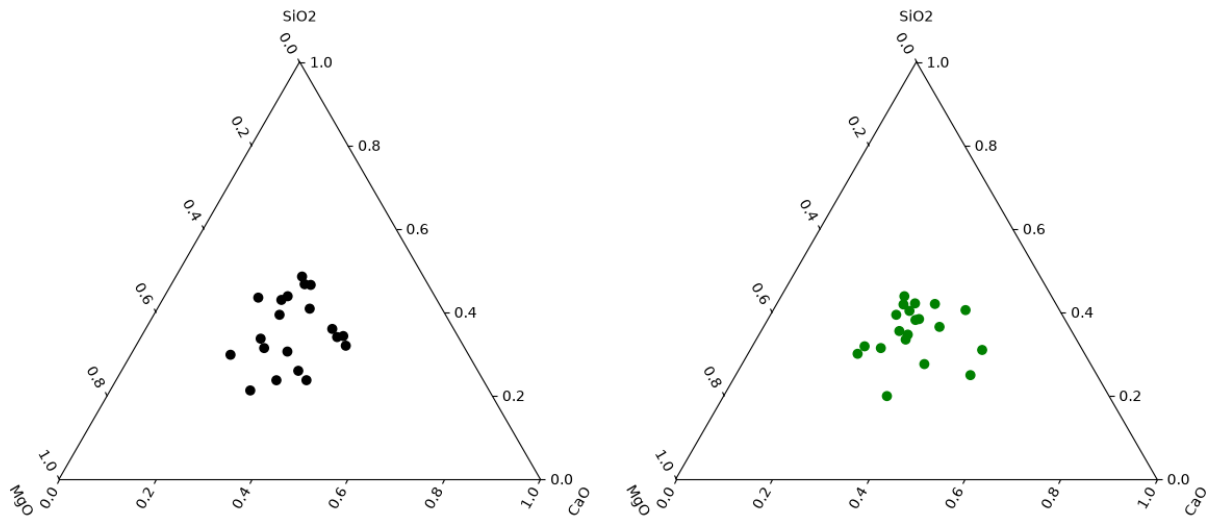


The plotting axis can be specified to use exisiting axes:

```
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(12, 5))

df.loc[:, ["SiO2", "MgO", "CaO"]].sample(20).pyroplot.scatter(ax=ax[0], c="k")
df.loc[:, ["SiO2", "MgO", "CaO"]].sample(20).pyroplot.scatter(ax=ax[1], c="g")

ax = fig.orderedaxes  # creating scatter plots reorders axes, this is the correct order
plt.tight_layout()
```



See also

Heatscatter Plots, Density Plots, Spider Density Diagrams, Ternary Color Mapping

Total running time of the script: (0 minutes 1.334 seconds)

4.4.3 REE Radii Plots

```
import matplotlib.pyplot as plt
```

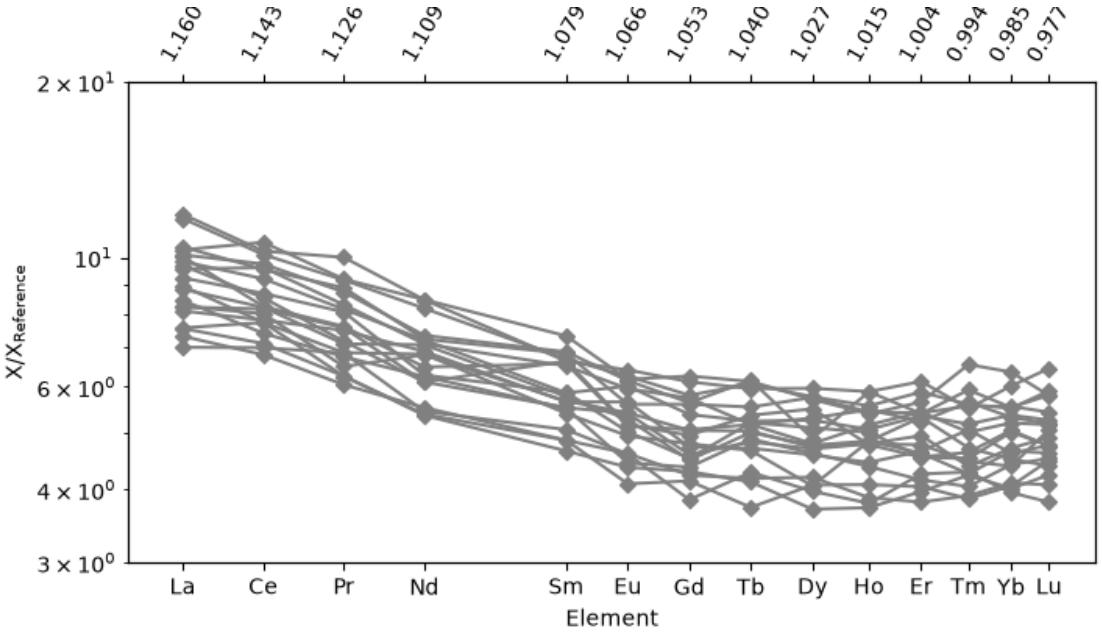
Here we generate some example data, using the *example_spider_data()* function (based on EMORB, here normalised to Primitive Mantle);

```
from pyrolite.util.synthetic import example_spider_data

df = example_spider_data(noise_level=0.1, size=20)
```

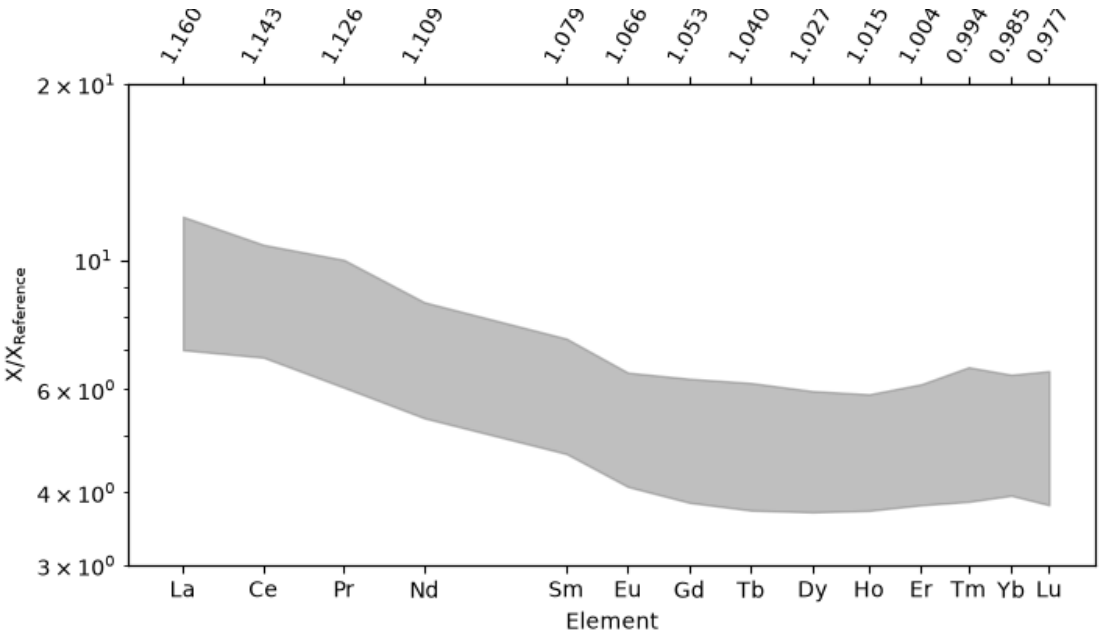
Where data is specified, the default plot is a line-based spiderplot:

```
ax = df.pyroplot.REE(color="0.5", figsize=(8, 4))
plt.show()
```



This behaviour can be modified (see spiderplot docs) to provide e.g. filled ranges:

```
df.pyroplot.REE(mode="fill", color="0.5", alpha=0.5, figsize=(8, 4))
plt.show()
```



The plotting axis can be specified to use exisiting axes:

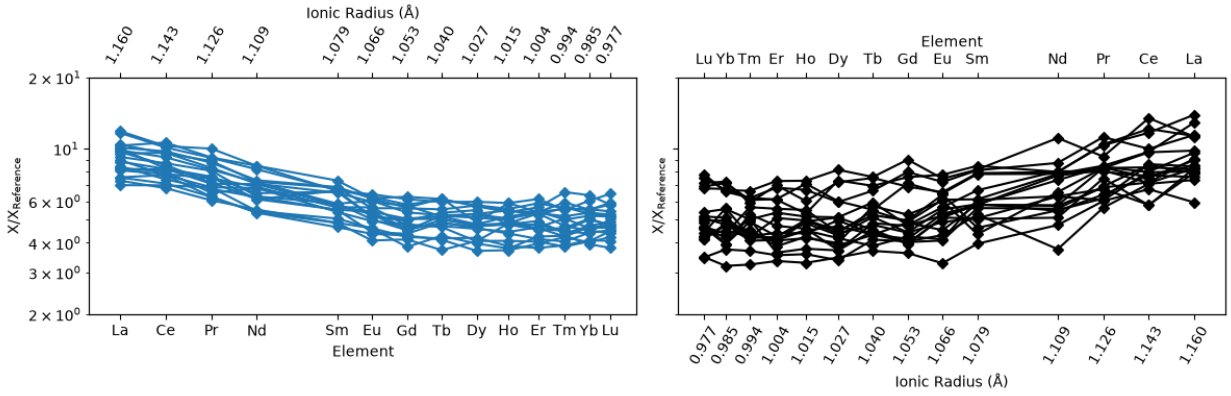
```
fig, ax = plt.subplots(1, 2, sharey=True, figsize=(12, 4))

df.pyroplot.REE(ax=ax[0])
# we can also change the index of the second axes
another_df = example_spider_data(noise_level=0.2, size=20)  # some 'nosier' data
another_df.pyroplot.REE(ax=ax[1], color="k", index="radii")
```

(continues on next page)

(continued from previous page)

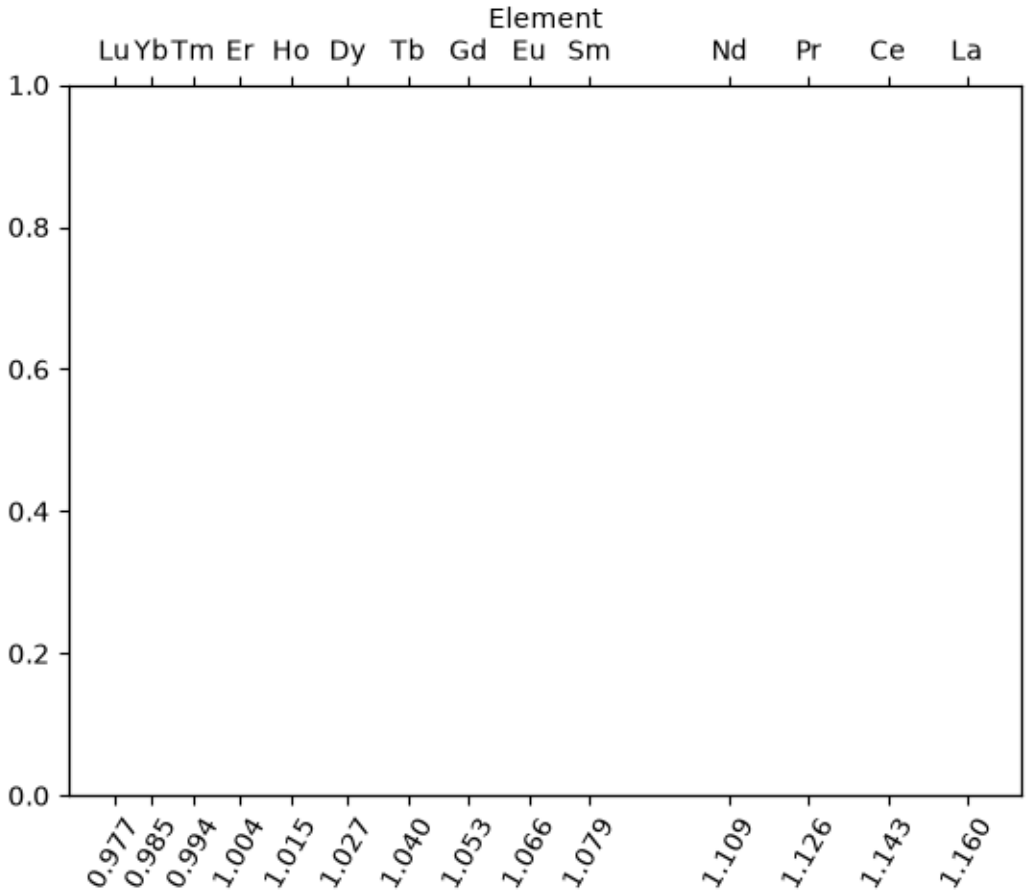
```
plt.tight_layout()
plt.show()
```



If you're just after a plotting template, you can use *REE_v_radii()* to get a formatted axis which can be used for subsequent plotting:

```
from pyrolite.plot.spider import REE_v_radii

ax = REE_v_radii(index="radii")  # radii mode will put ionic radii on the x axis
plt.show()
```



See also

Examples:
Ionic Radii, Spider Diagrams, lambdas: Parameterising REE Profiles

Functions:
get_ionic_radii(), *pyrolite.plot.pyroplot.REE()*, *pyrolite.plot.pyroplot.spider()*, *lambda_lnREE()*

Total running time of the script: (0 minutes 0.763 seconds)

4.4.4 Ternary Color Systems

`pyrolite` includes two methods for coloring data points and polygons in a ternary system, *ternary_color()* and *color_ternary_polygons_by_centroid()* which work well with some of the plot templates (*pyrolite.plot.templates*) and associated classifiers (*pyrolite.util.classification*).

```
import matplotlib.pyplot as plt
import numpy as np
```

Colors by Ternary Position

The *ternary_color()* function serves to generate the color which correpsonds the the mixing of three colours in proportion to the components in a ternary system. By default these colours are red, green and blue (corresponding to the top, left, and right components in a terarny diagram). The function returns colours in the form of an RGBA array:

```
from pyrolite.util.plot.style import ternary_color
from pyrolite.util.synthetic import normal_frame

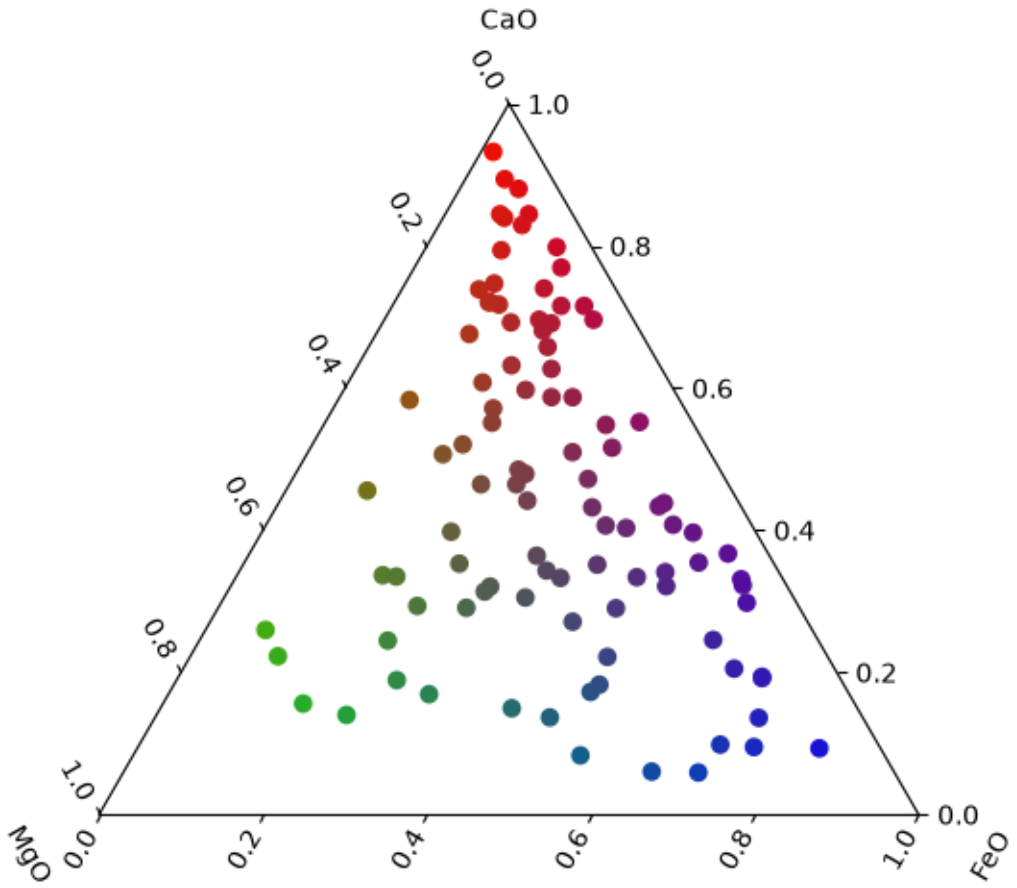
# generate a synthetic dataset we can use for the colouring example
df = normal_frame(
    columns=["CaO", "MgO", "FeO"],
    size=100,
    seed=42,
    cov=np.array([[0.8, 0.3], [0.3, 0.8]]),
)

colors = ternary_color(df)
colors[:3]
```

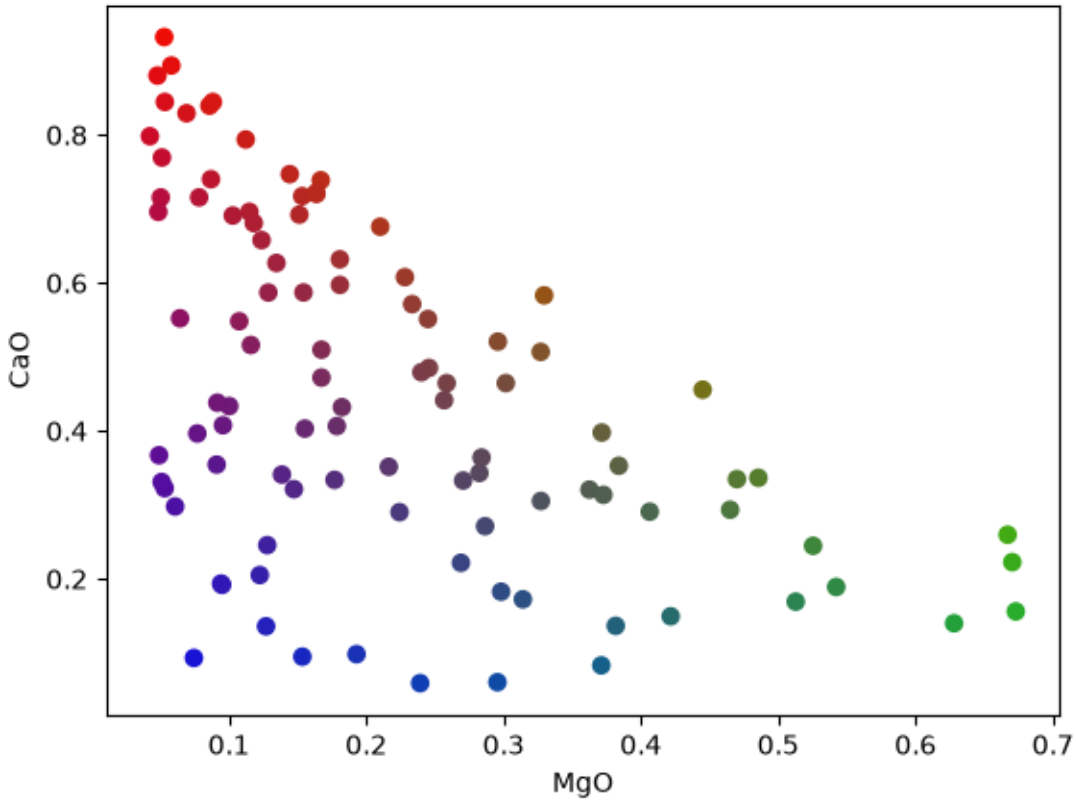
```
array([[0.18902861, 0.54222697, 0.26874443, 0.999999  ],
       [0.5097609 , 0.16739226, 0.32284684, 0.999999  ],
       [0.08310658, 0.37115312, 0.54574029, 0.999999  ]])
```

These can then be readily used in a ternary diagram (or eleswhere):

```
ax = df.pyroplot.scatter(c=colors)
plt.show()
```

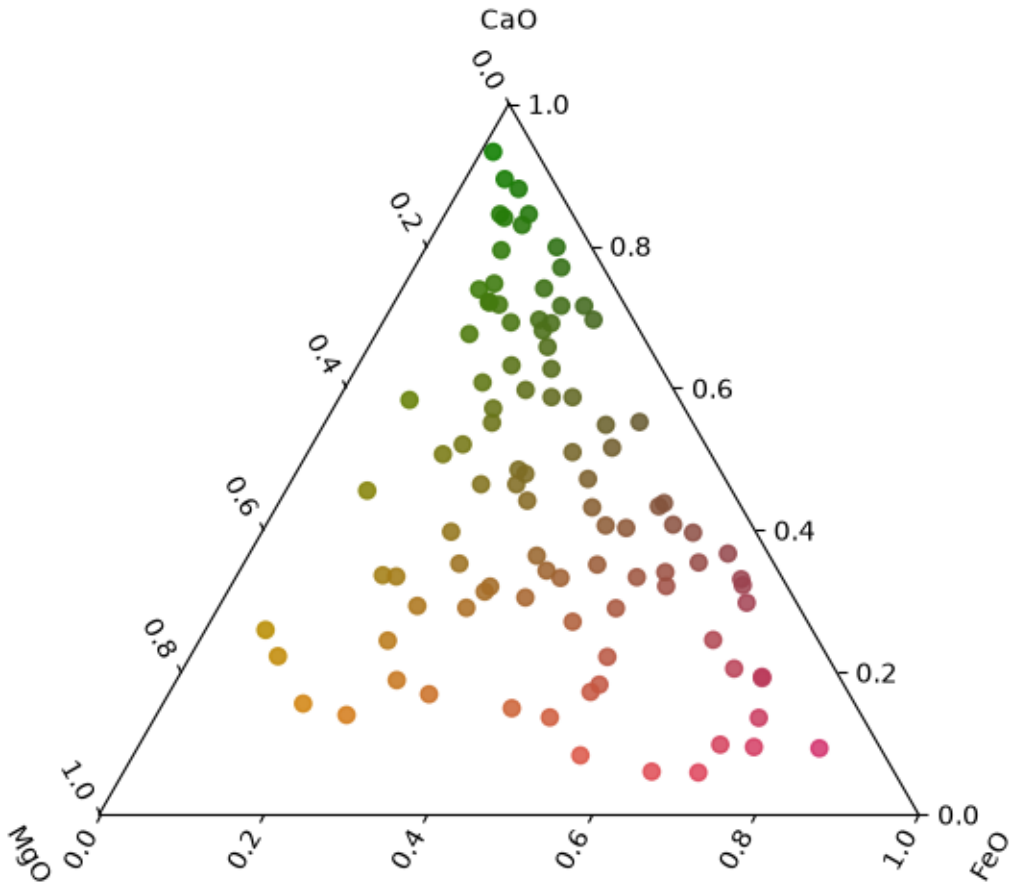



```
ax = df[["MgO", "CaO"]].pyroplot.scatter(c=colors)
plt.show()
```



You can use different colors for each of the verticies if you so wish, and mix and match named colors with RGB/RGBA represntations (note that the alpha will be scaled, if it is passed as a keyword argument to *ternary_color()*):

```
colors = ternary_color(df, alpha=0.9, colors=["green", "orange", [0.9, 0.1, 0.5, 0.9]])
ax = df.pyroplot.scatter(c=colors)
plt.show()
```

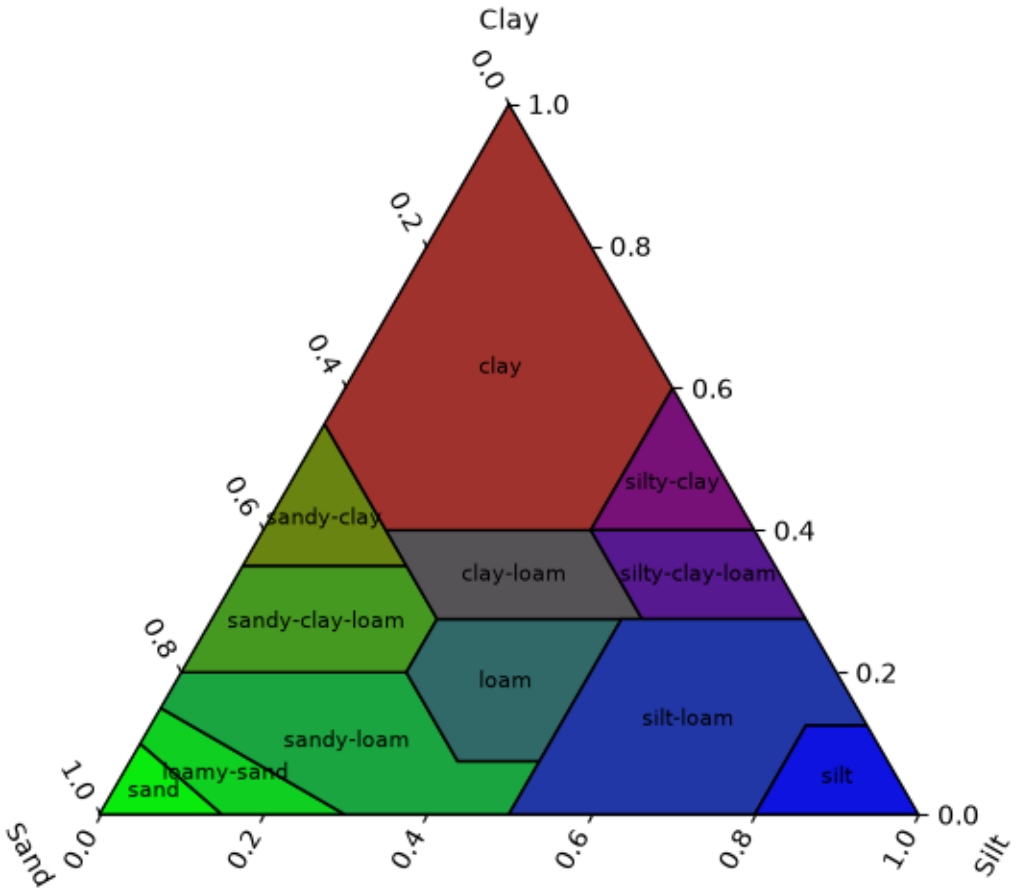


Colors by Centroid Position

We can also colour polygons within one of these templates by the ternary combination of colours (defaulting to red, green and blue) at the polygon centroid:

```
from pyrolite.util.classification import USDASoilTexture
from pyrolite.util.plot.style import color_ternary_polygons_by_centroid

clf = USDASoilTexture()
ax = clf.add_to_axes(ax=None, add_labels=True, figsize=(8, 8))
color_ternary_polygons_by_centroid(ax)
plt.show()
```




```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:56: UserWarning:␣
↪Non-positive entries found. Closure operation assumes all positive entries.
  warnings.warn(
```

There are a range of options you can pass to this function to control the ternary colors (as above), change the scaling coefficients for ternary components and change the opacity of the colors:

```
color_ternary_polygons_by_centroid(
    ax, colors=("red", "green", "blue"), coefficients=(1, 1, 1), alpha=0.5
)
plt.show()
```

See also

Examples:
Ternary Diagrams, Plot Templates

Total running time of the script: (0 minutes 0.630 seconds)

4.4.5 Parallel Coordinate Plots

Parallel coordinate plots are one way to visualise data relationships and clusters in higher dimensional data. pyrolite now includes an implementation of this which allows a handy quick exploratory visualisation.

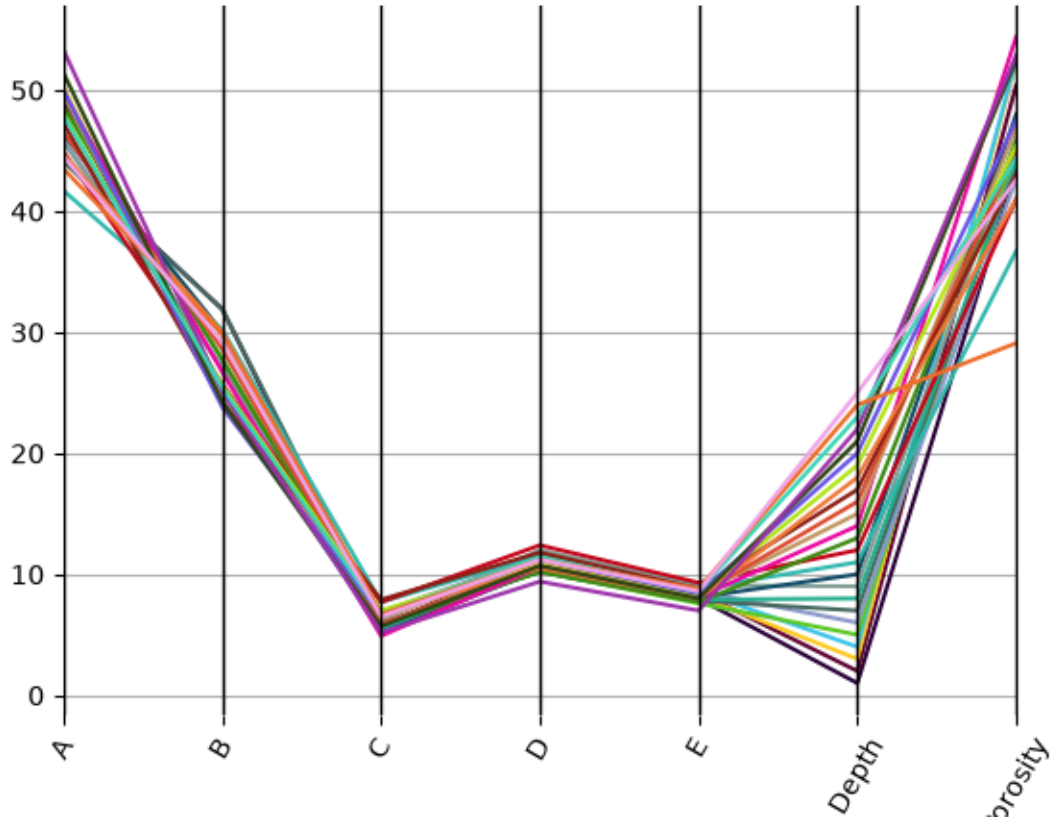
```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

import pyrolite.data.Aitchison
import pyrolite.plot
```

To start, let's load up an example dataset from Aitchison

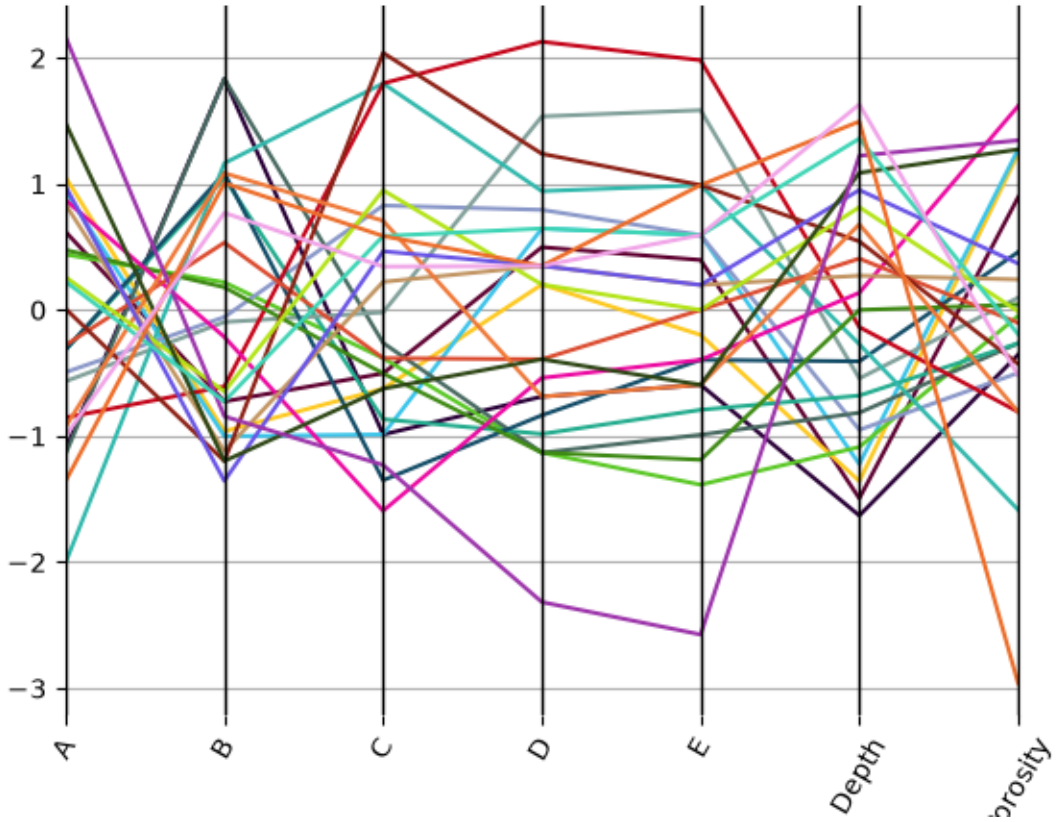
```
df = pyrolite.data.Aitchison.load_coxite()
comp = [
    i for i in df.columns if i not in ["Depth", "Porosity"]
]  # compositional data variables
```

```
ax = df.pyroplot.parallel()
plt.show()
```



By rescaling this using the mean and standard deviation, we can account for scale differences between variables:

```
ax = df.pyroplot.parallel(rescale=True)
plt.show()
```

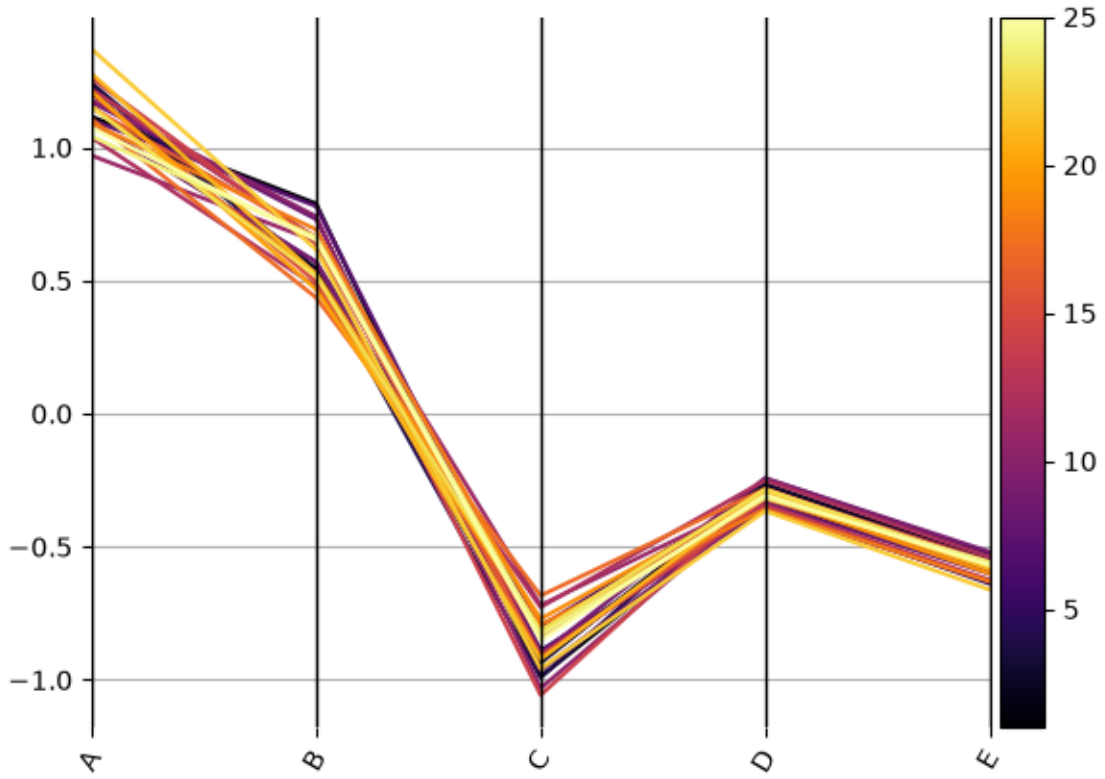


We can also use a centred-log transform for compositional data to reduce the effects of spurious correlation:

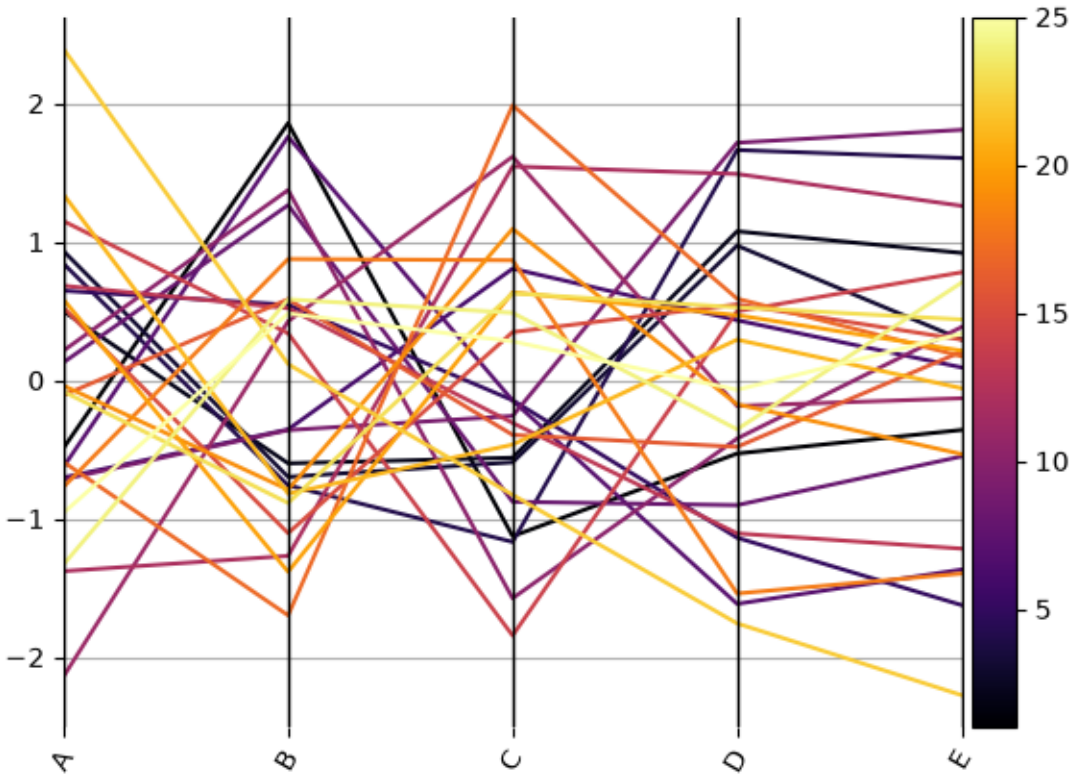
```
from pyrolite.util.skl.transform import CLRTransform

cmap = "inferno"
compdata = df.copy()
compdata[comp] = CLRTransform().transform(compdata[comp])
ax = compdata.loc[:, comp].pyroplot.parallel(color=compdata.Depth.values, cmap=cmap)
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)

# we can add a meaningful colorbar to indicate one variable also, here Depth
sm = plt.cm.ScalarMappable(cmap=cmap)
sm.set_array(df.Depth)
plt.colorbar(sm, cax=cax, orientation="vertical")
plt.show()
```




```
ax = compdata.loc[:, comp].pyroplot.parallel(
    rescale=True, color=compdata.Depth.values, cmap=cmap
)
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(sm, cax=cax, orientation="vertical")
plt.show()
```



Total running time of the script: (0 minutes 0.302 seconds)

4.4.6 Heatscatter Plots

While *density()* plots are useful summary visualizations for large datasets, scatterplots are more precise and retain all spatial information (although they can get crowded).

A scatter plot where individual points are coloured by data density in some respects represents the best of both worlds. A version inspired by similar existing visualisations is implemented with *heatscatter()*.

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(12)
```

First we'll create some example data

```
from pyrolite.util.synthetic import normal_frame, random_cov_matrix

df = normal_frame(
    size=1000,
    cov=random_cov_matrix(sigmas=np.random.rand(4) * 2, dim=4, seed=12),
    seed=12,
)
```

We can compare a minimal *heatscatter()* plot to other visualisations for the same data:

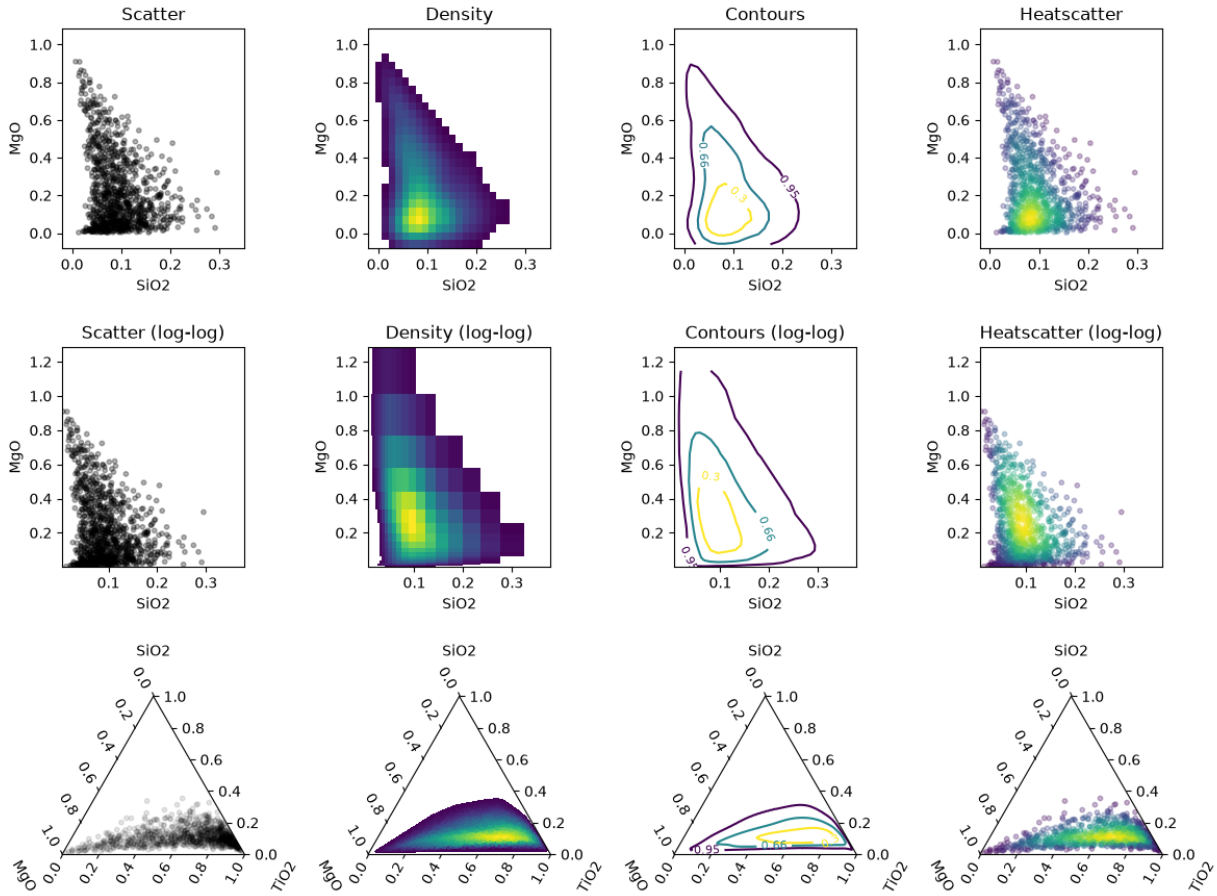
```
from pyrolite.util.plot.axes import share_axes

fig, ax = plt.subplots(3, 4, figsize=(12, 9))

ax = ax.flat
share_axes(ax[:4], which="xy")
share_axes(ax[4:8], which="xy")
share_axes(ax[8:], which="xy")

contours = [0.95, 0.66, 0.3]
bivar = ["SiO2", "MgO"]
trivar = ["SiO2", "MgO", "TiO2"]
# linear-scaled comparison
df.loc[:, bivar].pyroplot.scatter(ax=ax[0], c="k", s=10, alpha=0.3)
df.loc[:, bivar].pyroplot.density(ax=ax[1])
df.loc[:, bivar].pyroplot.density(ax=ax[2], contours=contours)
df.loc[:, bivar].pyroplot.heatscatter(ax=ax[3], s=10, alpha=0.3)
# log-log plots
df.loc[:, bivar].pyroplot.scatter(ax=ax[4], c="k", s=10, alpha=0.3)
df.loc[:, bivar].pyroplot.density(ax=ax[5], logx=True, logy=True)
df.loc[:, bivar].pyroplot.density(ax=ax[6], contours=contours, logx=True, logy=True)
df.loc[:, bivar].pyroplot.heatscatter(ax=ax[7], s=10, alpha=0.3, logx=True, logy=True)
# ternary plots
df.loc[:, trivar].pyroplot.scatter(ax=ax[8], c="k", s=10, alpha=0.1)
df.loc[:, trivar].pyroplot.density(ax=ax[9], bins=100)
df.loc[:, trivar].pyroplot.density(ax=ax[10], contours=contours, bins=100)
df.loc[:, trivar].pyroplot.heatscatter(ax=ax[11], s=10, alpha=0.3, renorm=True)
fig.subplots_adjust(hspace=0.4, wspace=0.4)

titles = ["Scatter", "Density", "Contours", "Heatscatter"]
for t, a in zip(titles + [i + " (log-log)" for i in titles], ax):
    a.set_title(t)
plt.tight_layout()
```



See also

Ternary Plots, Density Plots, Spider Density Diagrams

Total running time of the script: (0 minutes 2.696 seconds)

4.4.7 Plot Templates

`pyrolite` includes some ready-made templates for well-known plots. These can be used to create new plots, or add a template to an existing `matplotlib.axes.Axes`.

```
import matplotlib.pyplot as plt
```

Bivariate Templates

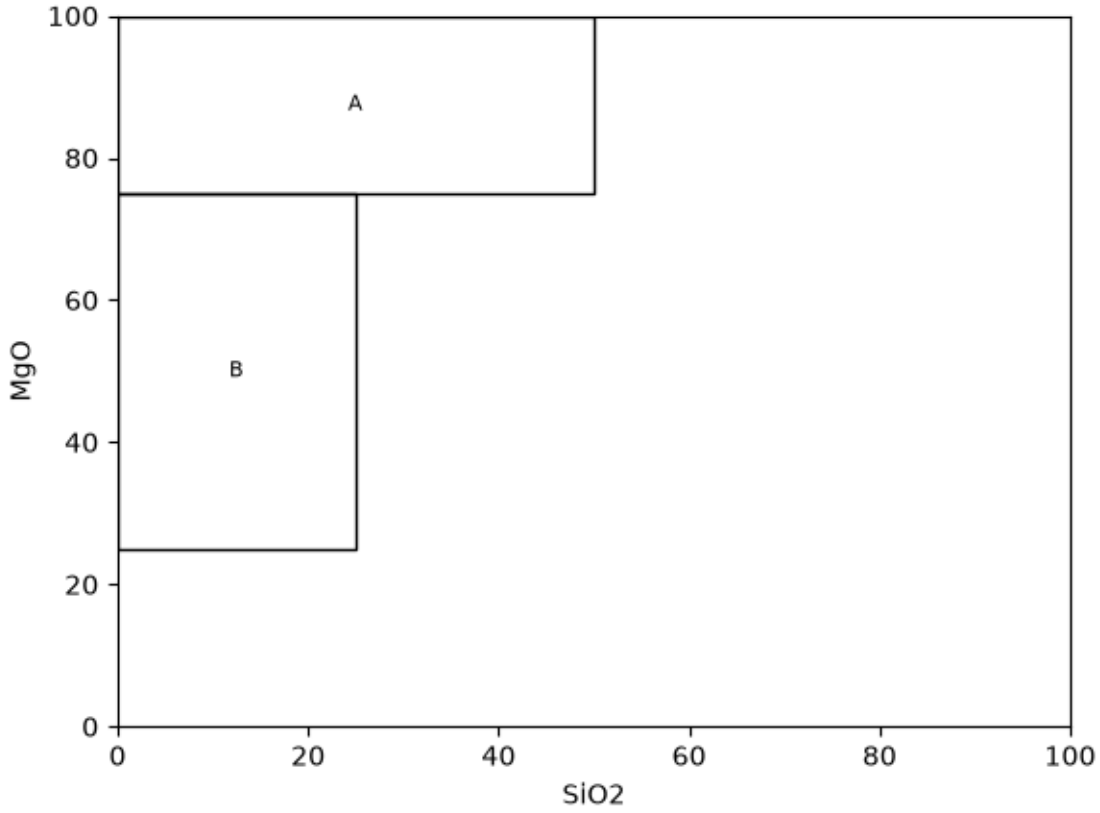
First let's build a simple total-alkali vs silica (*TAS()*) diagram:

```
from pyrolite.plot.templates import TAS, SpinelFeBivariate
from pyrolite.util.plot.axes import share_axes
```

(continues on next page)

(continued from previous page)

```
ax = TAS(linewidth=0.5, add_labels=True)
plt.show()
```

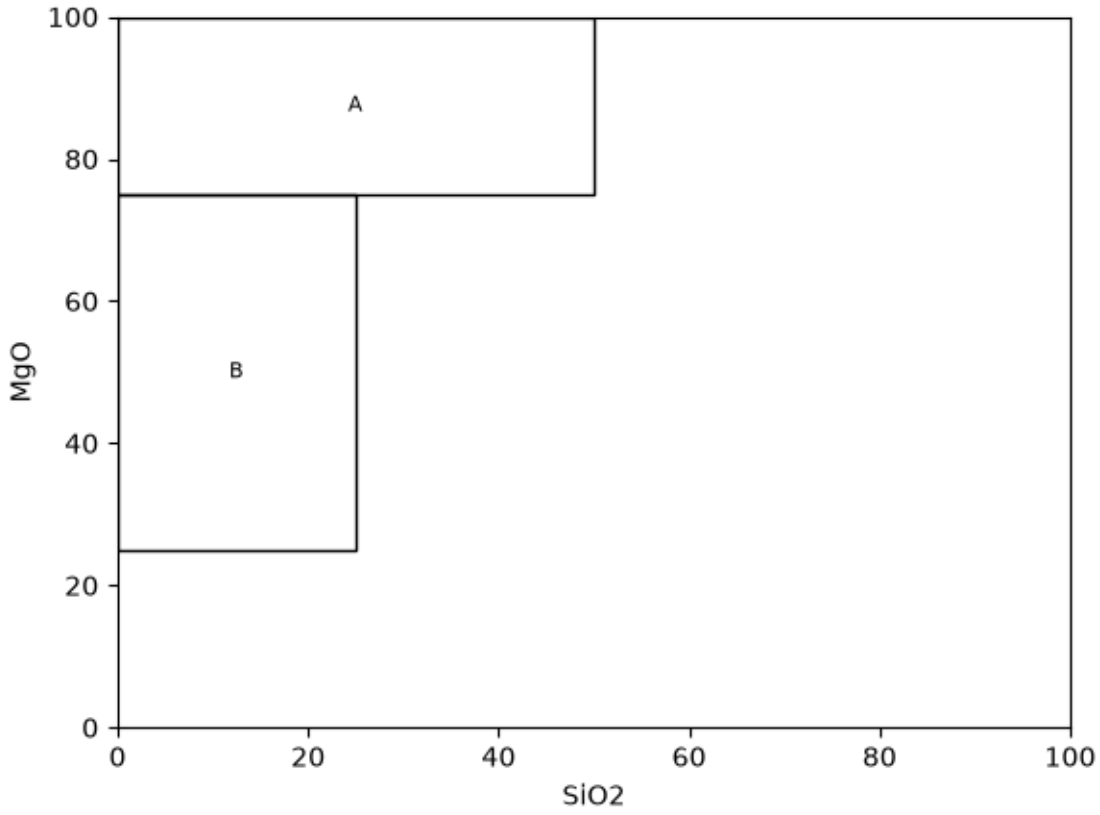


A few different variants are now available, with slightly different positioning of field boundaries, and with some fields combined:

```
fig, ax = plt.subplots(1, 3, figsize=(12, 3))

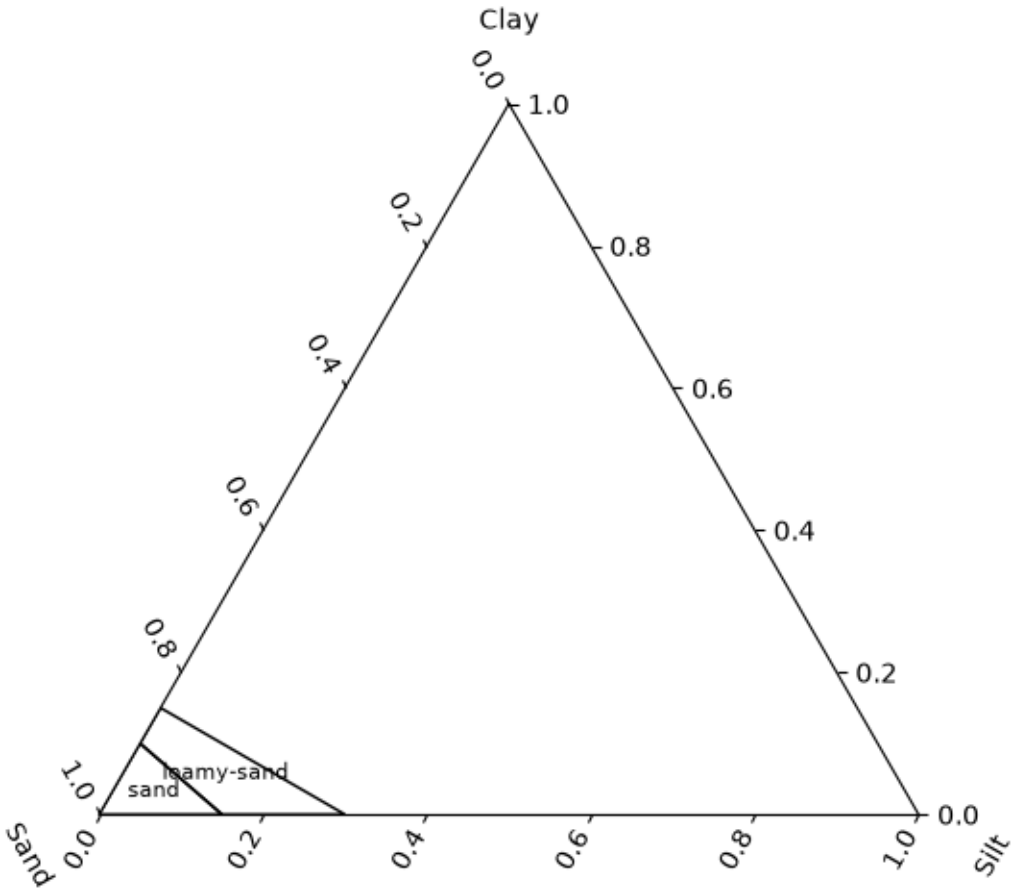
TAS(ax=ax[0], linewidth=0.5, add_labels=True, which_model=None)  # Middlemost's TAS
TAS(ax=ax[1], linewidth=0.5, add_labels=True, which_model="LeMaitre")  # LeMaitre's TAS
TAS(ax=ax[2], linewidth=0.5, add_labels=True, which_model="LeMaitreCombined")

for a in ax[1:]:
    a.set(yticks=[], ylabel=None)
```



For distinguishing Fe-rich variants of spinel phases, the bivariate spinel diagram can be useful:

```
ax = SpinelFeBivariate(linewidth=0.5, add_labels=True)
plt.show()
```



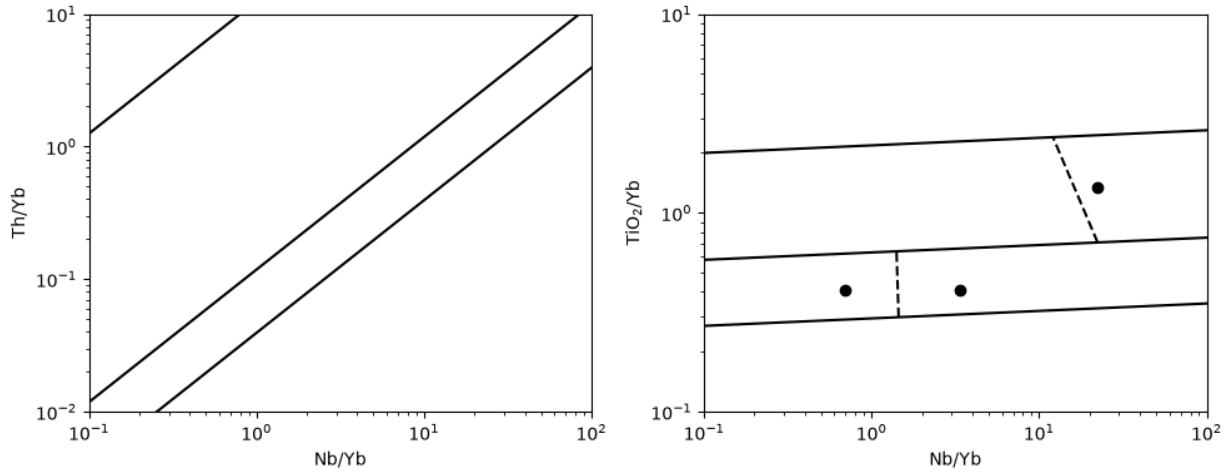
pyrolite contains templates for the Pearce diagrams, used to discriminate mafic rocks (and particularly basalts) based on their whole-rock geochemistry. Two templates are included: *pearceThNbYb()* and *pearceTiNbYb()*. We can create some axes and add these templates to them:

```
from pyrolite.plot.templates import pearceThNbYb, pearceTiNbYb

fig, ax = plt.subplots(1, 2, figsize=(10, 4))
share_axes(ax, which="x")  # these diagrams have the same x axis

pearceThNbYb(ax=ax[0])
pearceTiNbYb(ax=ax[1])

plt.tight_layout()  # nicer spacing for axis labels
```



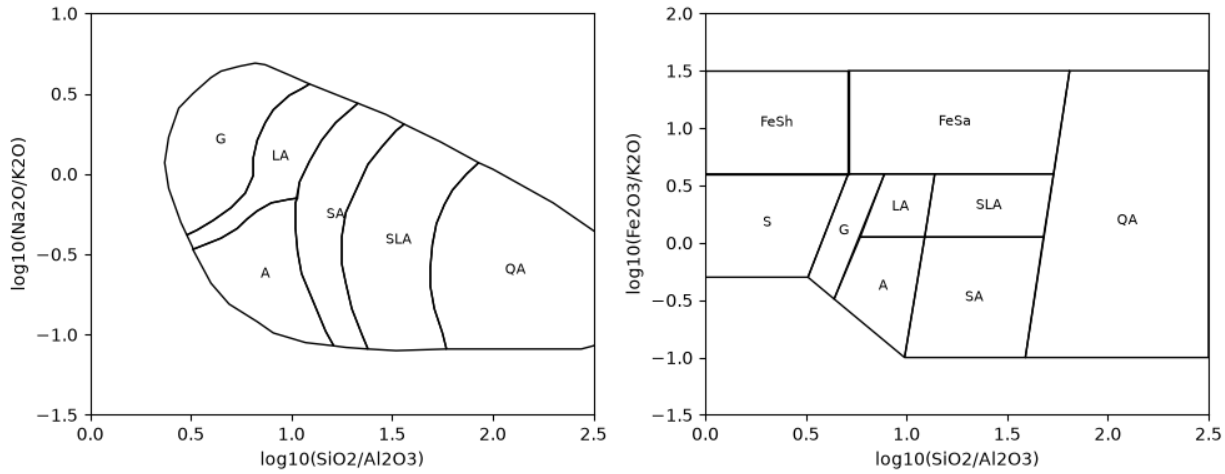
pyrolite also now includes some diagram templates for discrimination of sandstones based on their whole-rock geochemistry (*Pettijohn()*, *Herron()*):

```
from pyrolite.plot.templates import Herron, Pettijohn

fig, ax = plt.subplots(1, 2, figsize=(10, 4))
share_axes(ax, which="x")  # these diagrams have the same x axis

Pettijohn(ax=ax[0], add_labels=True)
Herron(ax=ax[1], add_labels=True)

plt.tight_layout()
```



Ternary Templates

pyrolite now also includes ternary classification diagrams inlcuding the *QAP()* and *USDASoilTexture()* diagrams:

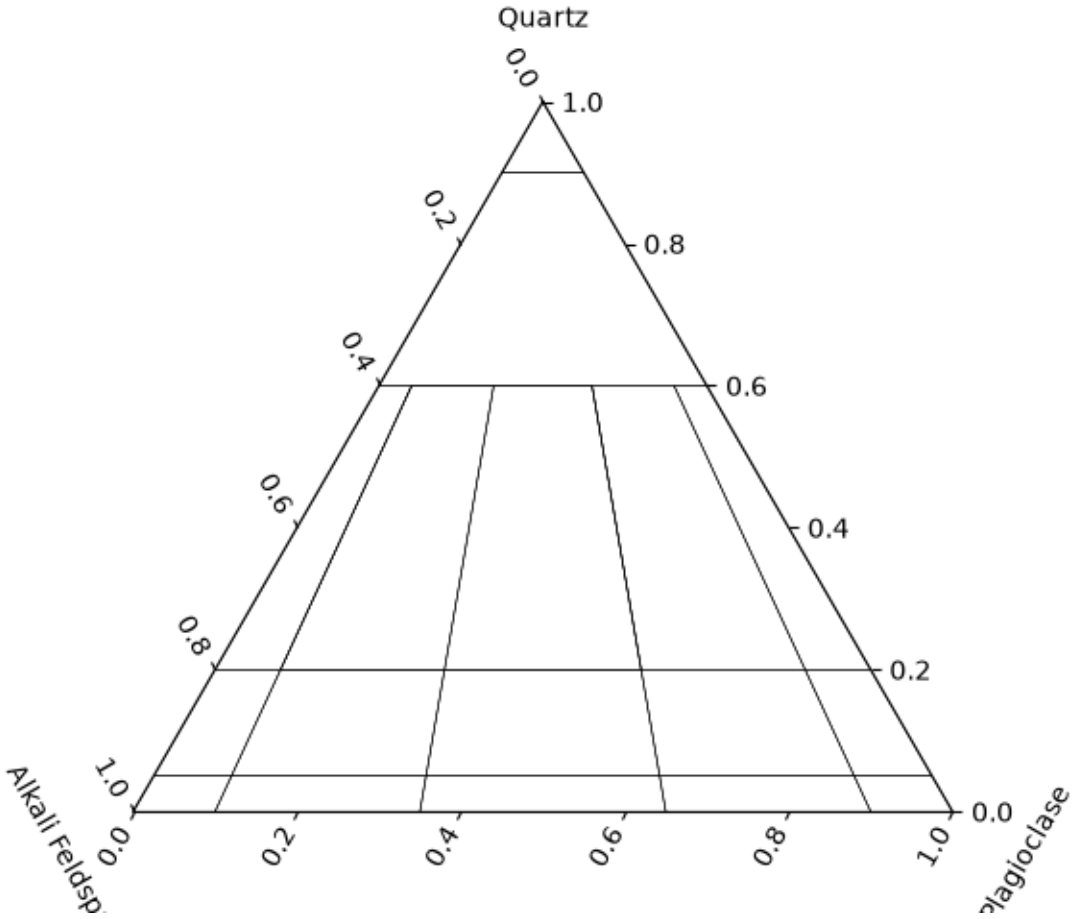
```
from pyrolite.plot.templates import (
    QAP,
    FeldsparTernary,
    JensenPlot,
```

(continues on next page)

(continued from previous page)

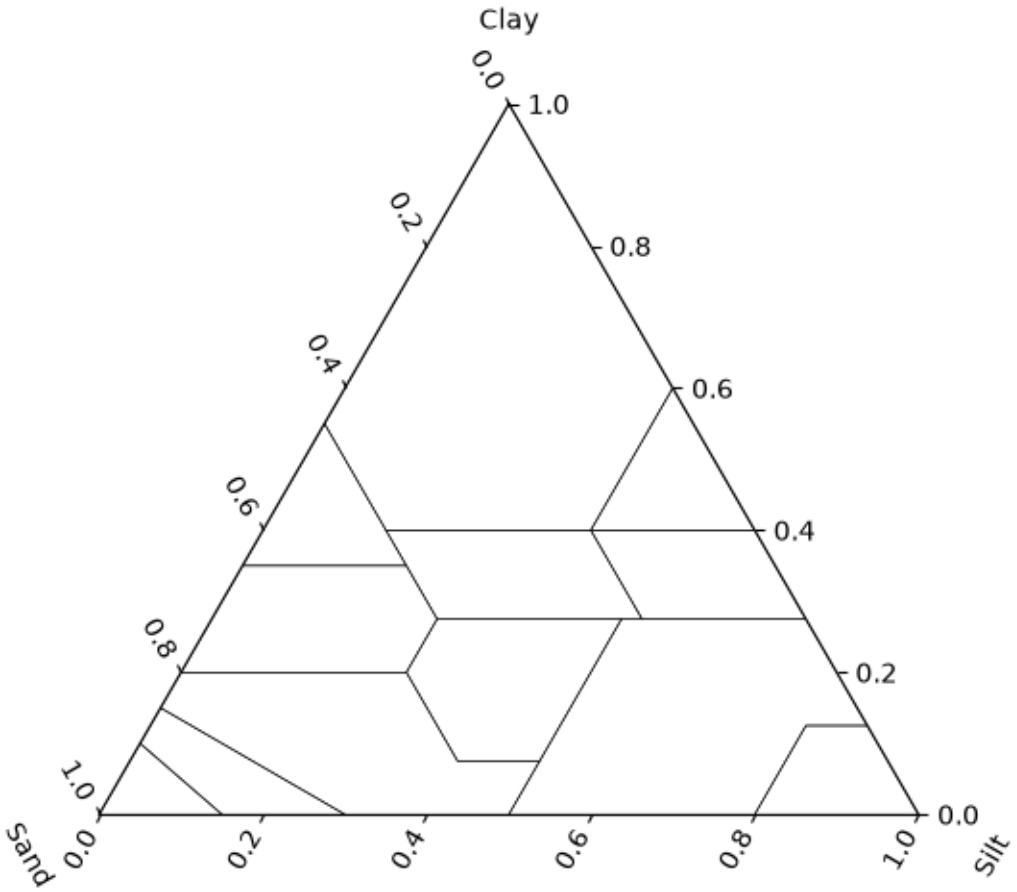
```
    SpinelTrivalentTernary,
    USDASoilTexture,
)

ax = QAP(linewidth=0.4)
plt.show()
```



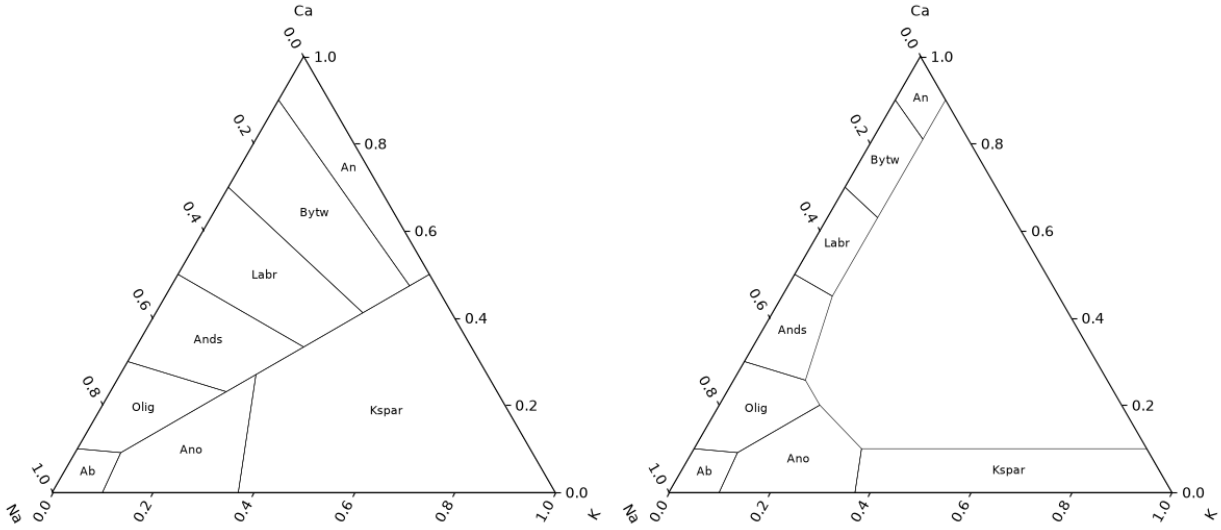

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:56: UserWarning:␣
↪Non-positive entries found. Closure operation assumes all positive entries.
  warnings.warn(
```

```
ax = USDASoilTexture(linewidth=0.4)
plt.show()
```



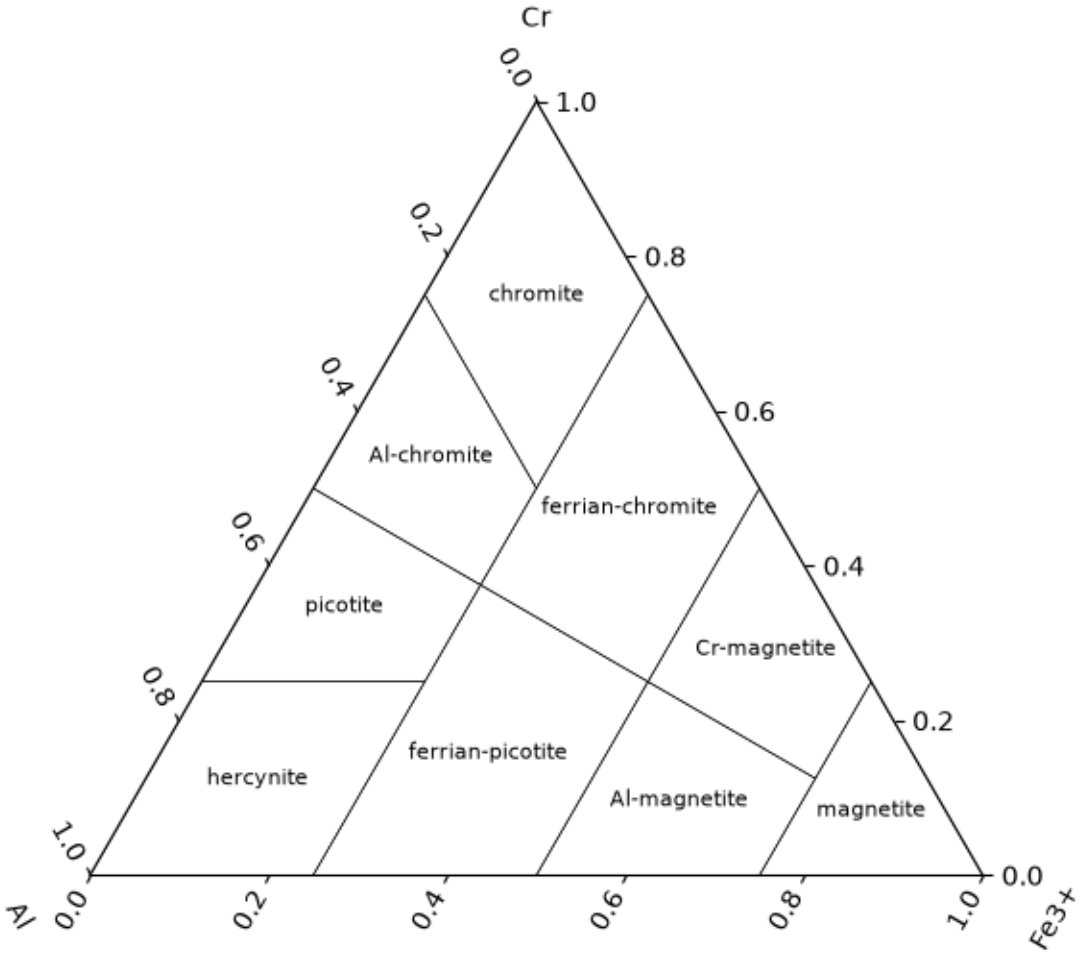
For the feldspar ternary diagram, which is complicated by a miscibility gap, there are two modes: *'default'* and *'miscibility-gap'*. The second of these provides a simplified approximation of the miscibility gap between k-feldspar and plagioclase, wheras 'default' ignores this aspect (which itself is complicated by temperature):

```
fig, ax = plt.subplots(1, 2, figsize=(12, 6))
FeldsparTernary(ax=ax[0], linewidth=0.4, add_labels=True, mode="default")
FeldsparTernary(ax=ax[1], linewidth=0.4, add_labels=True, mode="miscibility-gap")
plt.tight_layout()
plt.show()
```



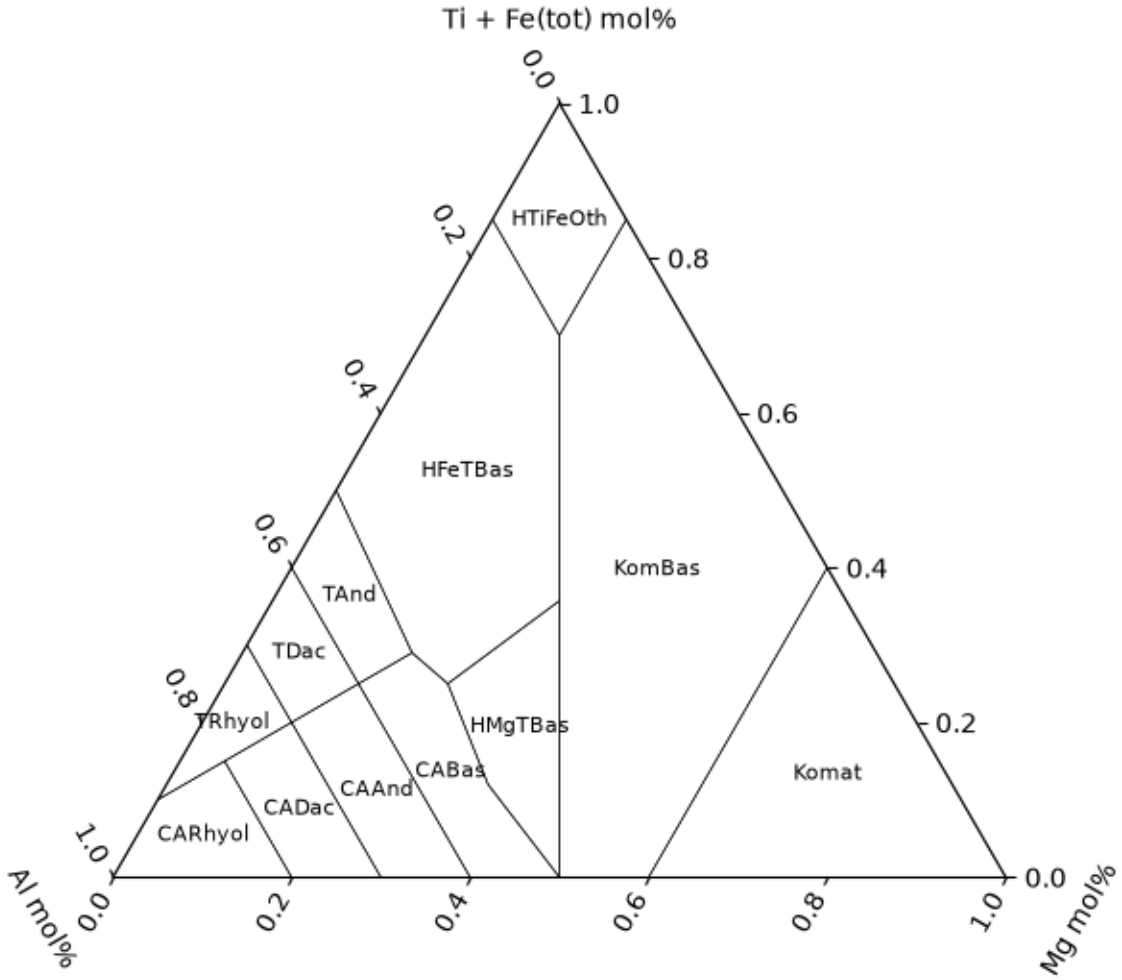
For general spinel phase discrimination, a ternary classification diagram can be used to give labels based on trivalent cationic content (Cr^{3+}, Al^{3+}, Fe^{3+}):

```
SpinelTrivalentTernary(linewidth=0.4, add_labels=True, figsize=(6, 6))
plt.show()
```



The Jensen plot is another cationic ternary discrimination diagram (Jensen, 1976), for subalkaline volcanic rocks:

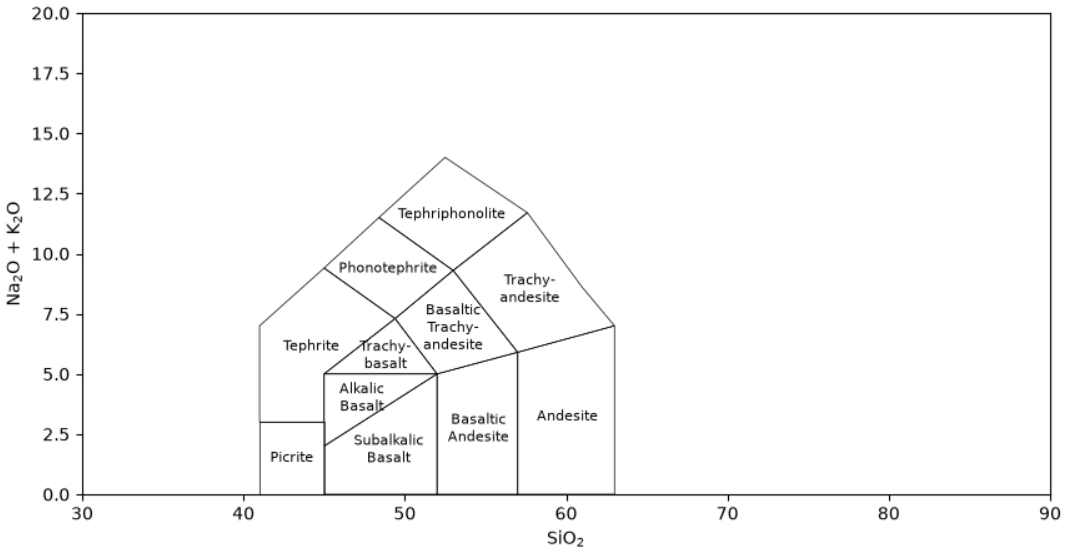
```
JensenPlot(linewidth=0.4, add_labels=True, figsize=(6, 6))
plt.show()
```



Customization - Labelling Schemes and Field Subsets

For most templates, you're able to customize which labels are applied to each field. This includes swapping between the 'ID' and the 'name' (and for TAS, swapping between intrusive and volcanic equivalents), as well as specifing that only fields with specific IDs are added to the diagram. For example, here we add volcanic labels for a subset of fields in a TAS diagram (in this case, the LeMaitre version):

```
ax = TAS(
    linewidth=0.5,
    add_labels=True,
    which_model="LeMaitre",
    which_labels="volcanic",
    which_ids=["Pc", "Ba", "Bs", "O1", "O2", "U1", "U2", "U3", "S1", "S2", "S3"],
    figsize=(10, 5),
)
ax.figure
```




```
<Figure size 1000x500 with 1 Axes>
```

References and other notes for diagram templates can be found within the docstrings and within the pyrolite documentation:

```
help(TAS)
```

```
Help on function TAS in module pyrolite.plot.templates.TAS:

TAS(
    ax: matplotlib.axes._axes.Axes | None = None,
    add_labels: bool = False,
    which_labels: str = 'ID',
    relim: bool = True,
    color: str = 'k',
    which_model: str | None = None,
    **kwargs
) -> matplotlib.axes._axes.Axes
    Adds the TAS diagram to an axes. Diagram from Middlemost (1994) [#pyrolite.plot.
↪templates.TAS.TAS_1]_,
    a closed-polygon variant after Le Bas et al (1992) [#pyrolite.plot.templates.TAS.TAS_
↪2]_.

    Parameters
    ----------
    ax : matplotlib.axes.Axes
        Axes to add the template on to.
    add_labels : bool
        Whether to add labels at polygon centroids.
    which_labels : str
        Which labels to add to the polygons (e.g. for TAS, 'volcanic', 'intrusive'
        or the field 'ID').
    relim : bool
```

(continues on next page)

(continued from previous page)

```
        Whether to relimit axes to fit the built in ranges for this diagram.
    color : str
        Line color for the diagram.
    which_model : str
        The name of the model variant to use, if not Middlemost.

    Returns
    -------
    ax : matplotlib.axes.Axes

    References
    ----------
    .. [#pyrolite.plot.templates.TAS.TAS_1] Middlemost, E. A. K. (1994).
                Naming materials in the magma/igneous rock system.
                Earth-Science Reviews, 37(3), 215-224.
                doi: `10.1016/0012-8252(94)90029-9 <https://dx.doi.org/10.1016/0012-
↪8252(94)90029-9>`__

    .. [#pyrolite.plot.templates.TAS.TAS_2] Le Bas, M.J., Le Maitre, R.W., Woolley, A.R.␣
↪(1992).
                The construction of the Total Alkali-Silica chemical
                classification of volcanic rocks.
                Mineralogy and Petrology 46, 1-22.
                doi: `10.1007/BF01160698 <https://dx.doi.org/10.1007/BF01160698>`__
```

See also

Examples:
TAS Classifier, Ternary Colour Mapping

Modules:
pyrolite.util.classification

Classes:
TAS, *QAP*, *USDASoilTexture*

Total running time of the script: (0 minutes 2.379 seconds)

4.4.8 Density and Contour Plots

While individual point data are useful, we commonly want to understand the the distribution of our data within a particular subspace, and compare that to a reference or other dataset. Pyrolite includes a few functions for visualising data density, most based on Gaussian kernel density estimation and evaluation over a grid. The below examples highlight some of the currently implemented features.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from pyrolite.comp.codata import close
```

(continues on next page)

(continued from previous page)

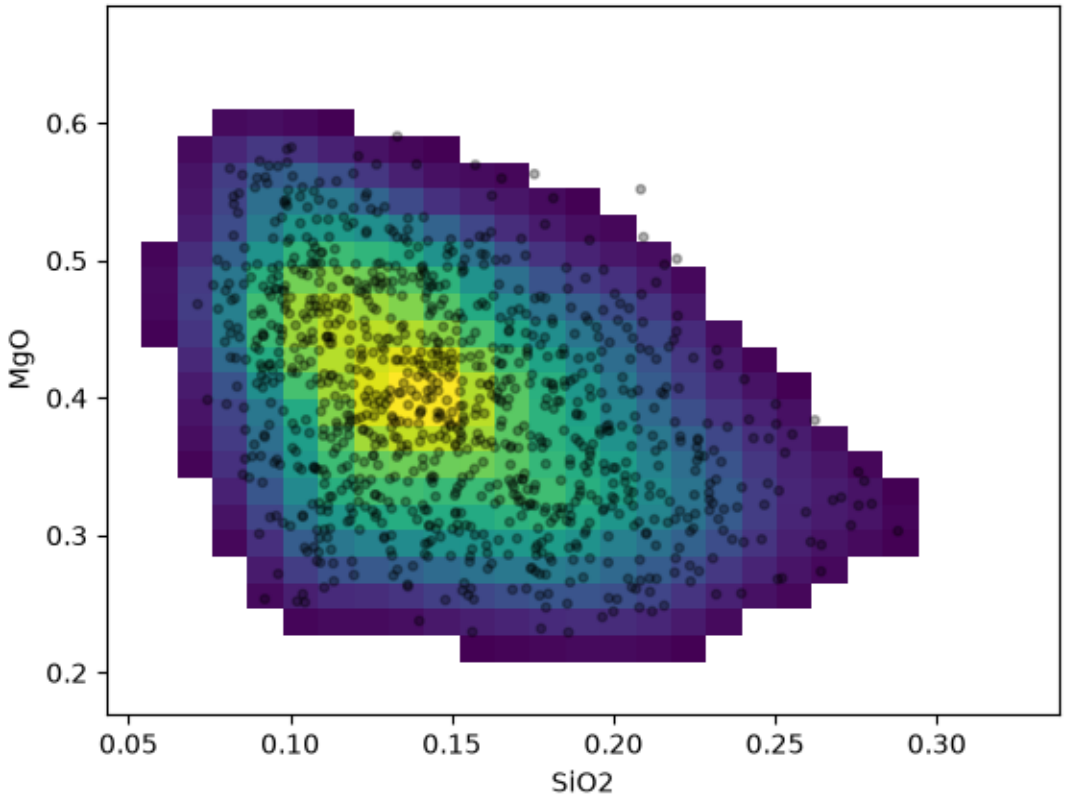
```
np.random.seed(82)
```

First we create some example data :

```
oxs = ["SiO2", "CaO", "MgO", "Na2O"]
ys = np.random.rand(1000, len(oxs))
ys[:, 1] += 0.7
ys[:, 2] += 1.0
df = pd.DataFrame(data=close(np.exp(ys)), columns=oxs)
```

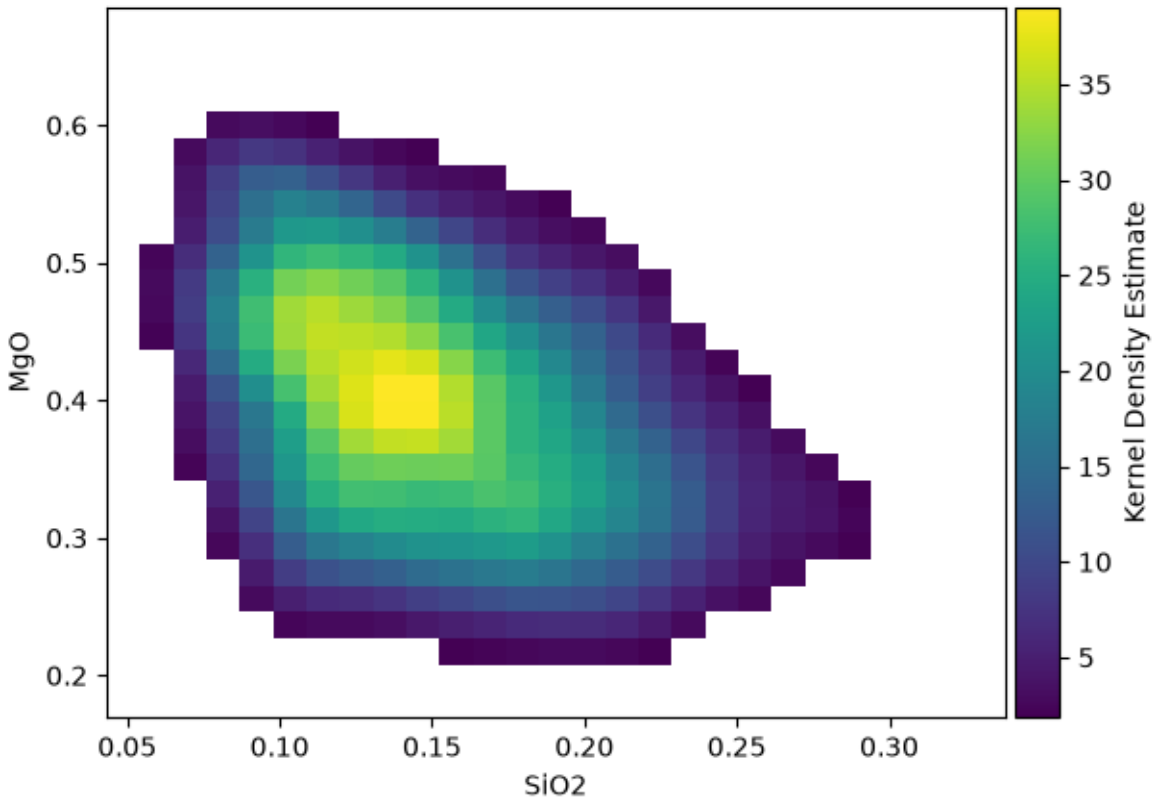
A minimal density plot can be constructed as follows:

```
ax = df.loc[:, ["SiO2", "MgO"]].pyroplot.density()
df.loc[:, ["SiO2", "MgO"]].pyroplot.scatter(ax=ax, s=10, alpha=0.3, c="k", zorder=2)
plt.show()
```



A colorbar linked to the KDE estimate colormap can be added using the *colorbar* boolean switch:

```
ax = df.loc[:, ["SiO2", "MgO"]].pyroplot.density(colorbar=True)
plt.show()
```

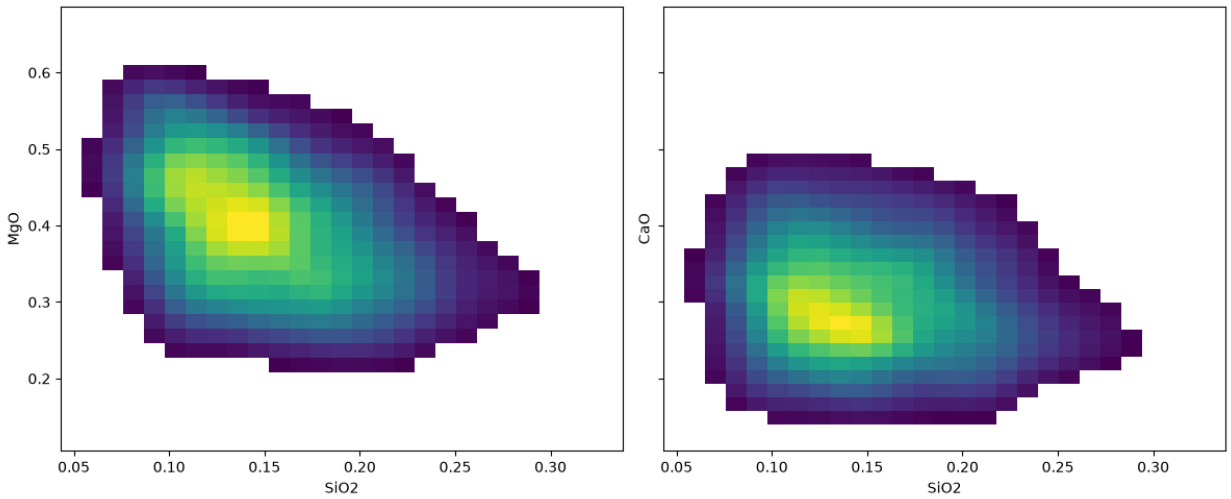


density by default will create a new axis, but can also be plotted over an existing axis for more control:

```
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(12, 5))

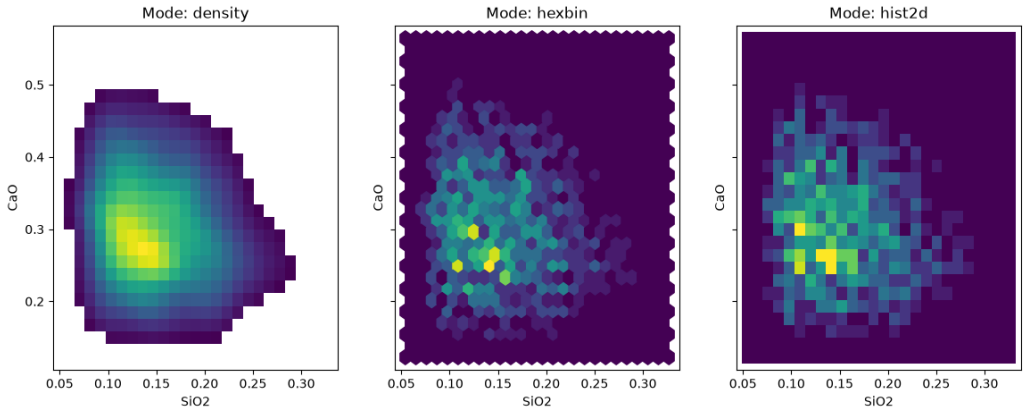
df.loc[:, ["SiO2", "MgO"]].pyroplot.density(ax=ax[0])
df.loc[:, ["SiO2", "CaO"]].pyroplot.density(ax=ax[1])

plt.tight_layout()
plt.show()
```



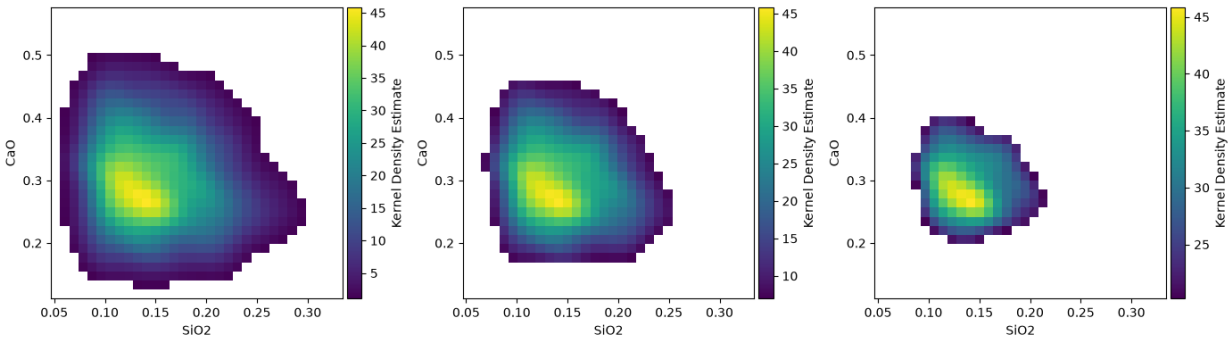
There are two other implemented modes beyond the default *density*: *hist2d* and *hexbin*, which parallel their equivalents in matplotlib. Contouring is not enabled for these histogram methods.

```
fig, ax = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(14, 5))
for a, mode in zip(ax, ["density", "hexbin", "hist2d"]):
    df.loc[:, ["SiO2", "CaO"]].pyroplot.density(ax=a, mode=mode)
    a.set_title(f"Mode: {mode}")
plt.show()
```



For the `density` mode, a `vmin` parameter is used to choose the lower threshold, and by default is the 99th percentile (`vmin=0.01`), but can be adjusted. This is useful where there are a number of outliers, or where you wish to reduce the overall complexity/colour intensity of a figure (also good for printing!).

```
fig, ax = plt.subplots(1, 3, figsize=(14, 4))
for a, vmin in zip(ax, [0.01, 0.1, 0.4]):
    df.loc[:, ["SiO2", "CaO"]].pyroplot.density(ax=a, bins=30, vmin=vmin, colorbar=True)
plt.tight_layout()
plt.show()
```

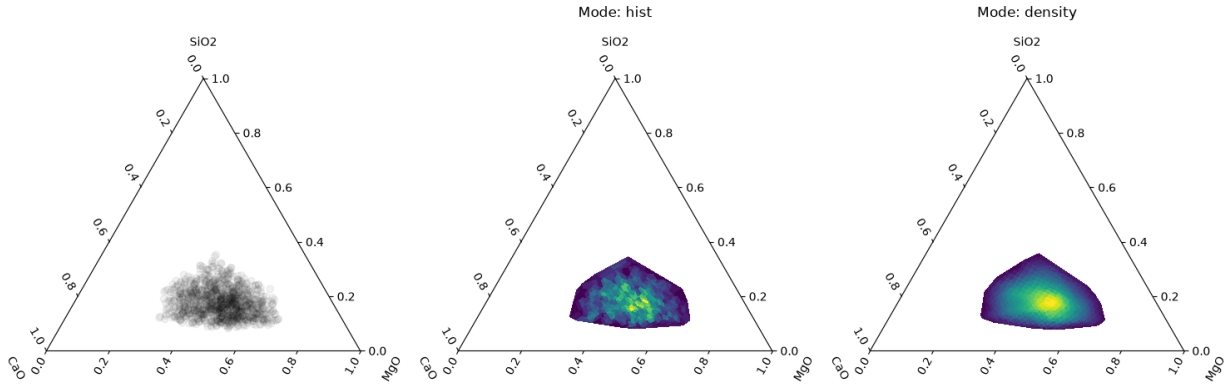



```
plt.close("all")  # let's save some memory..
```

Density plots can also be used for ternary diagrams, where more than two components are specified:

```
fig, ax = plt.subplots(
    1,
    3,
    sharex=True,
    sharey=True,
    figsize=(15, 5),
    subplot_kw=dict(projection="ternary"),
)
df.loc[:, ["SiO2", "CaO", "MgO"]].pyroplot.scatter(ax=ax[0], alpha=0.05, c="k")
for a, mode in zip(ax[1:], ["hist", "density"]):
    df.loc[:, ["SiO2", "CaO", "MgO"]].pyroplot.density(ax=a, mode=mode)
    a.set_title(f"Mode: {mode}", y=1.2)

plt.tight_layout()
plt.show()
```



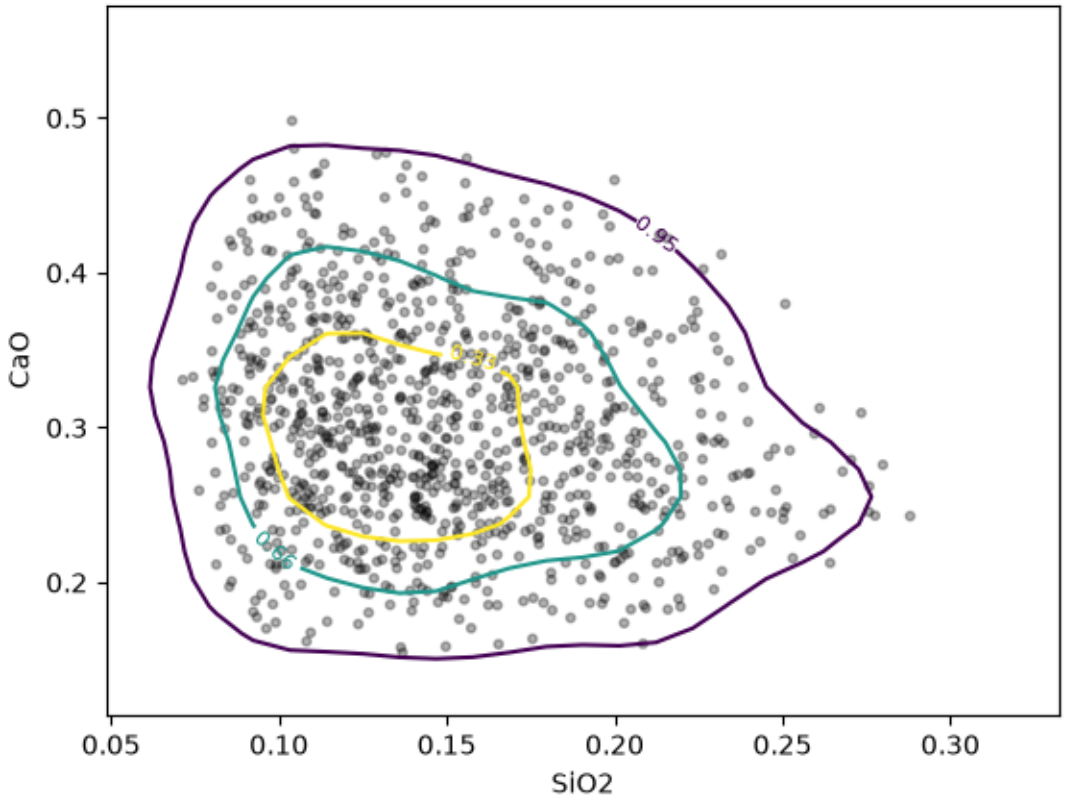
Contours are also easily created, which by default are percentile estimates of the underlying kernel density estimate.

Note

As the contours are generated from kernel density estimates, assumptions around e.g. 95% of points lying within a 95% contour won't necessarily be valid for non-normally distributed data (instead, this represents the approximate 95% percentile on the kernel density estimate). Note that contours are currently only generated; for

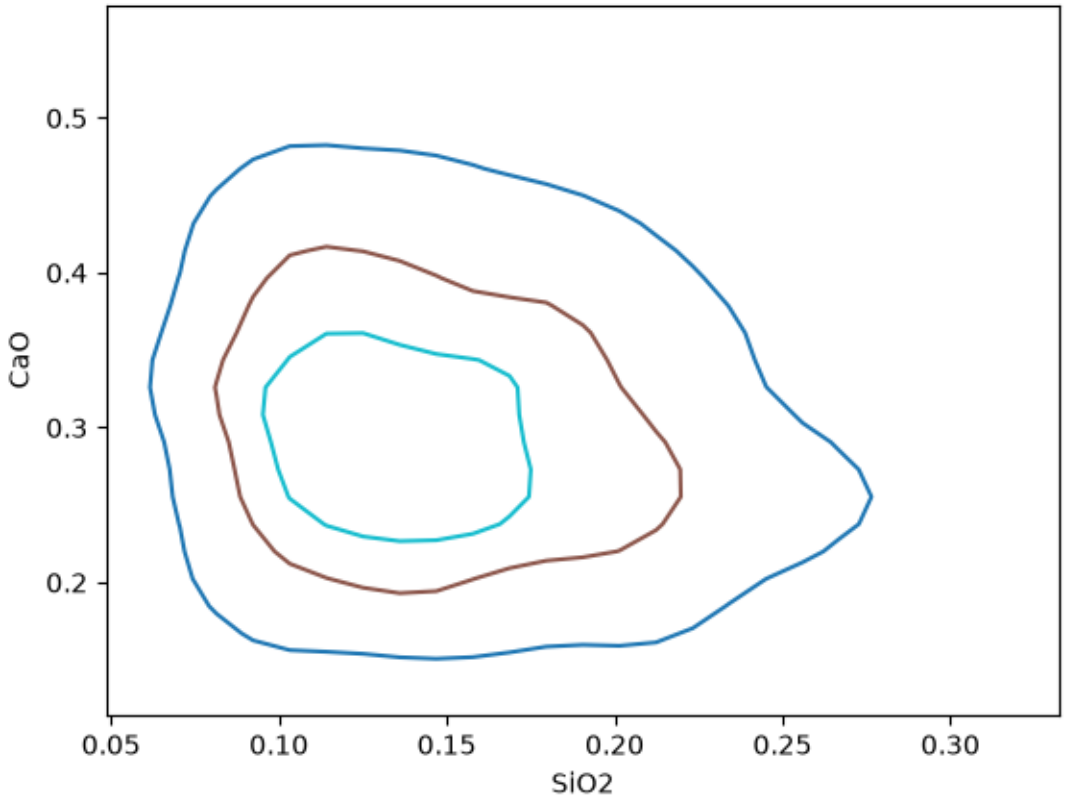
mode="density"; future updates may allow the use of a histogram basis, which would give results closer to 95% data percentiles.

```
ax = df.loc[:, ["SiO2", "CaO"]].pyroplot.scatter(s=10, alpha=0.3, c="k", zorder=2)
df.loc[:, ["SiO2", "CaO"]].pyroplot.density(ax=ax, contours=[0.95, 0.66, 0.33])
plt.show()
```



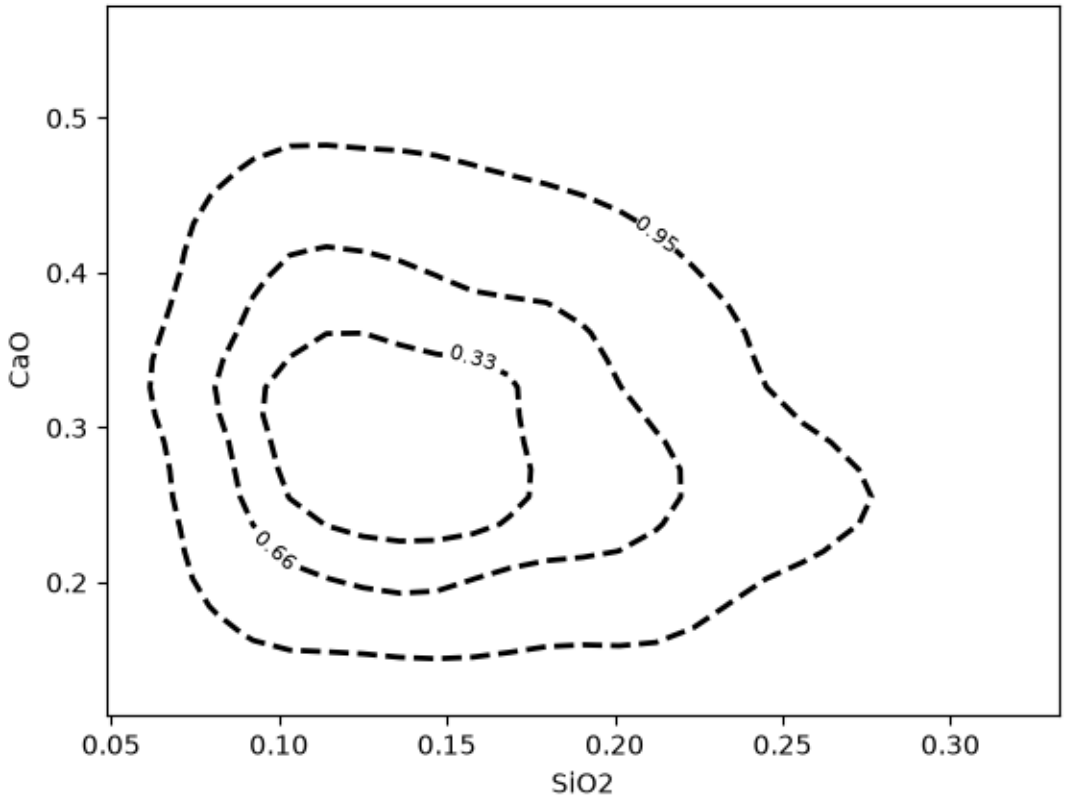
You can readily change the styling of these contours by passing *cmap* (for different colormaps), *colors* (for bulk or individually specified colors), *linewidths* and *linestyles* (for bulk or individually specified linestyles). You can also use 'label_contours=False` to omit the contour labels. For example, to change the colormap and omit the contour labels:

```
ax = df.loc[:, ["SiO2", "CaO"]].pyroplot.density(
    contours=[0.95, 0.66, 0.33], cmap="tab10", label_contours=False
)
plt.show()
```



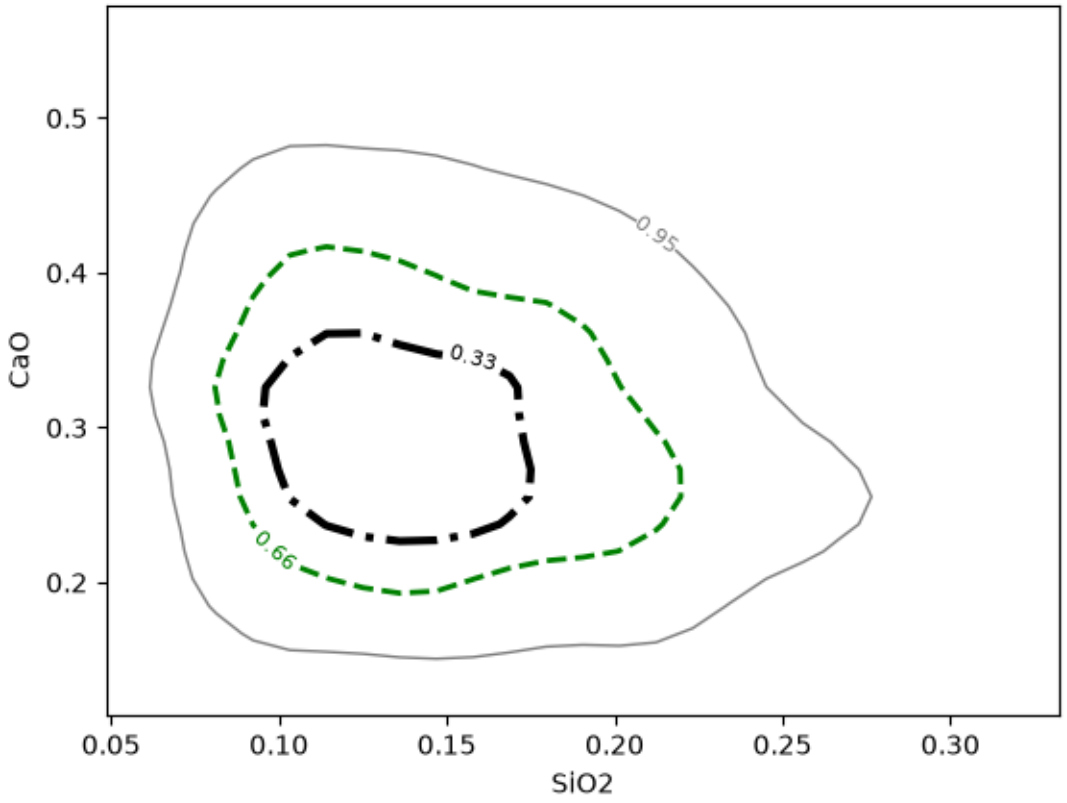
You can specify one styling value for all contours:

```
ax = df.loc[:, ["SiO2", "CaO"]].pyroplot.density(
    contours=[0.95, 0.66, 0.33], linewidths=2, linestyles="--", colors="k"
)
plt.show()
```



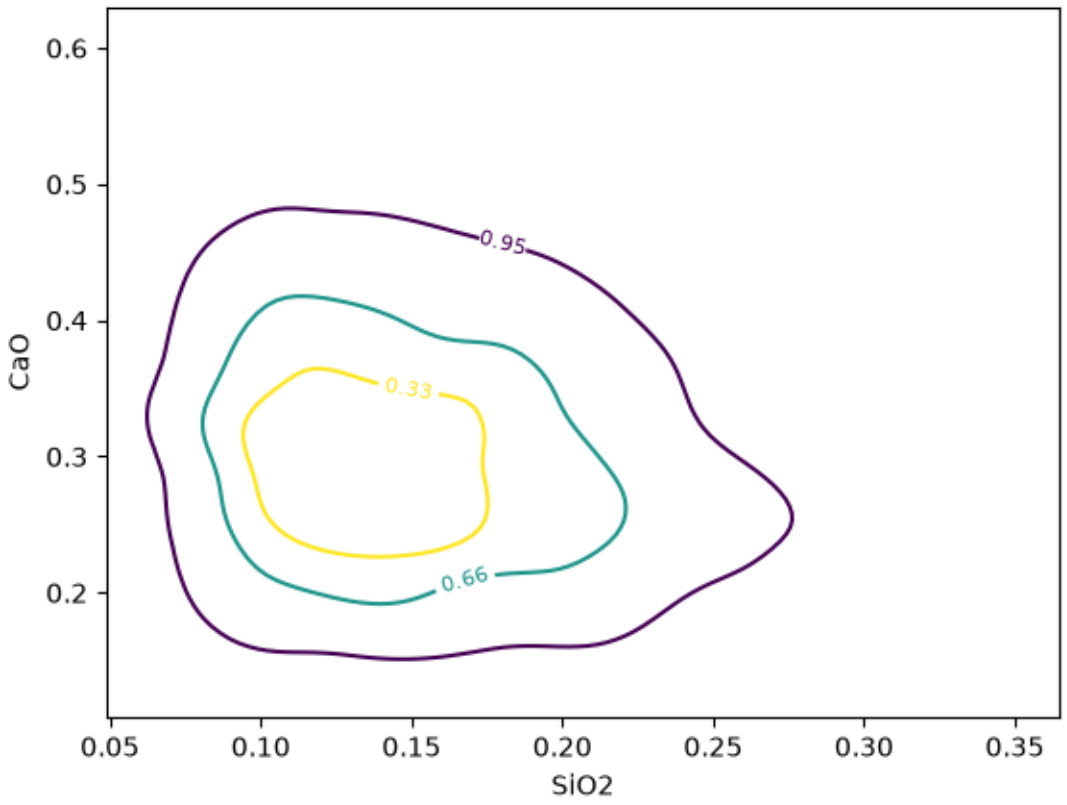
Or specify a list of values which will be applied to the contours in the order given in the *contours* keyword argument:

```
ax = df.loc[:, ["SiO2", "CaO"]].pyroplot.density(
    contours=[0.95, 0.66, 0.33],
    linewidths=[1, 2, 3],
    linestyles=["-", "--", "-."],
    colors=["0.5", "g", "k"],
)
plt.show()
```



If for some reason the density plot or contours are cut off on the edges, it may be that the span of the underlying grid doesn't cut it (this is often the case for just a small number of points). You can either manually adjust the extent (*extent = (min_x, max_x, min_y, max_y)*) or adjust the coverage scale (e.g. *coverage_scale = 1.5* will increase the buffer from the default 10% to 25%). You can also adjust the number of bins in the underlying grid (either for both axes with e.g. *bins=100* or individually with *bins=(<xbins>, <ybins>)*):

```
ax = df.loc[:, ["SiO2", "CaO"]].pyroplot.density(
    contours=[0.95, 0.66, 0.33], coverage_scale=1.5, bins=100
)
plt.show()
```



Geochemical data is commonly log-normally distributed and is best analysed and visualised after log-transformation. The density estimation can be conducted over logspaced grids (individually for x and y axes using *logx* and *logy* boolean switches). Notably, this makes both the KDE image and contours behave more naturally:

```
# some assymetric data
from scipy import stats

xs = stats.norm.rvs(loc=6, scale=3, size=(200, 1))
ys = stats.norm.rvs(loc=20, scale=3, size=(200, 1)) + 5 * xs + 50
data = np.append(xs, ys, axis=1).T
asym_df = pd.DataFrame(np.exp(np.append(xs, ys, axis=1) / 25.0))
asym_df.columns = ["A", "B"]
grids = ["linxy", "logxy"] * 2 + ["logx", "logy"]
scales = ["linscale"] * 2 + ["logscale"] * 2 + ["semilogx", "semilogy"]
labels = [f"{ls}-{ps}" for (ls, ps) in zip(grids, scales)]
params = list(
    zip(
        [
            (False, False),
            (True, True),
            (False, False),
            (True, True),
            (True, False),
            (False, True),
```

(continues on next page)

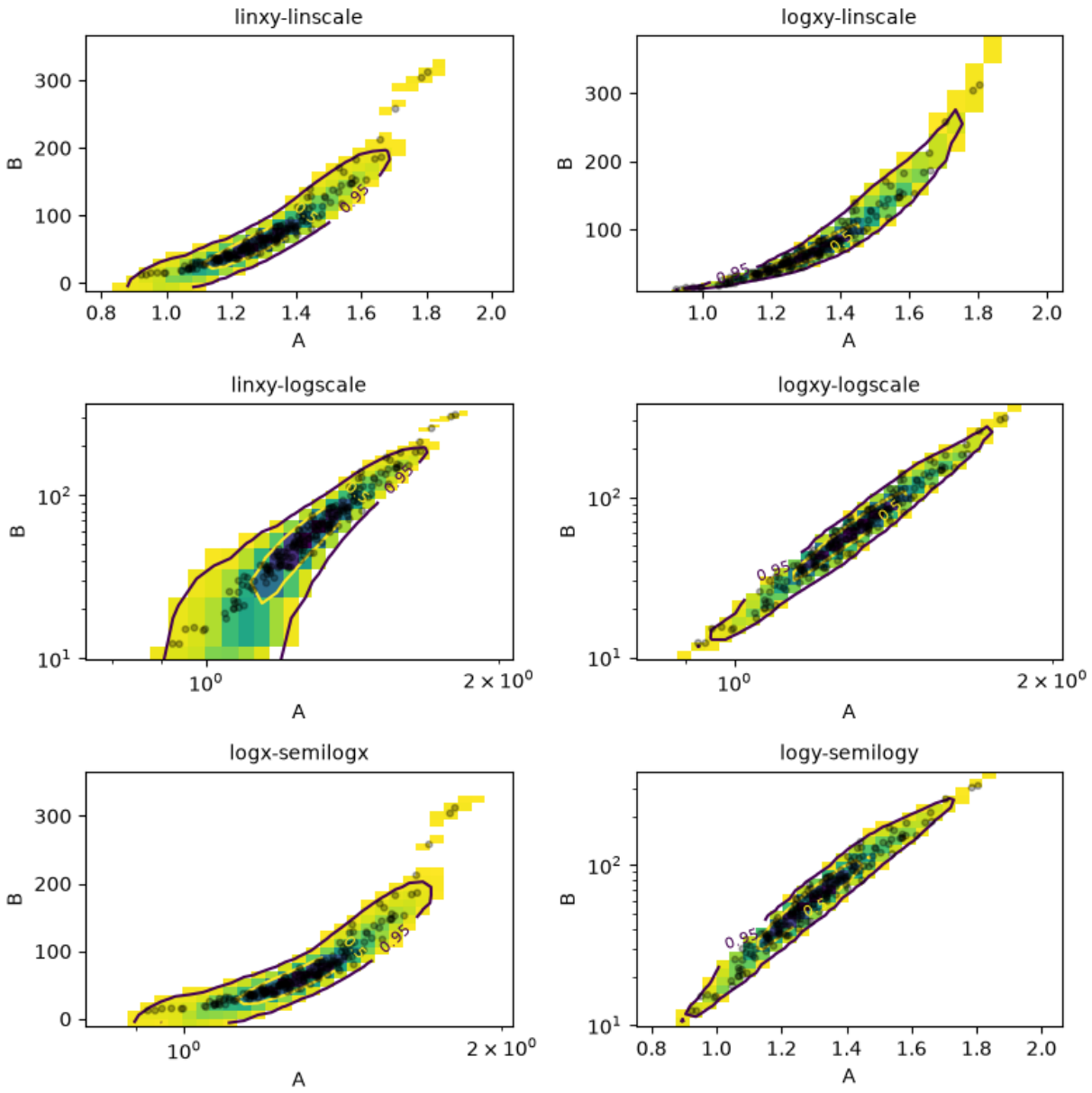
(continued from previous page)

```
        ],
        grids,
        scales,
    )
)
```

```
fig, ax = plt.subplots(3, 2, figsize=(8, 8))
ax = ax.flat

for a, (ls, grid, scale) in zip(ax, params):
    lx, ly = ls
    asym_df.pyroplot.density(ax=a, logx=lx, logy=ly, bins=30, cmap="viridis_r")
    asym_df.pyroplot.density(
        ax=a,
        logx=lx,
        logy=ly,
        contours=[0.95, 0.5],
        bins=30,
        cmap="viridis",
        fontsize=10,
    )
    asym_df.pyroplot.scatter(ax=a, s=10, alpha=0.3, c="k", zorder=2)

    a.set_title(f"{grid}-{scale}", fontsize=10)
    if scale in ["logscale", "semilogx"]:
        a.set_xscale("log")
    if scale in ["logscale", "semilogy"]:
        a.set_yscale("log")
plt.tight_layout()
plt.show()
```




```
plt.close("all")  # let's save some memory..
```

Note

Using alpha with the `density` mode induces a known and old matplotlib bug, where the edges of bins within a `pcolormesh` image (used for plotting the KDE estimate) are over-emphasized, giving a gridded look.

See also

Heatscatter Plots, Ternary Plots, Spider Density Diagrams

Total running time of the script: (0 minutes 2.400 seconds)

4.4.9 Spiderplots & Density Spiderplots

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Here we'll set up an example which uses EMORB as a starting point. Typically we'll normalise trace element compositions to a reference composition to be able to link the diagram to 'relative enrichement' occuring during geological processes, so here we're normalising to a Primitive Mantle composition first. We're here taking this normalised composition and adding some noise in log-space to generate multiple compositions about this mean (i.e. a compositional distribution). For simplicility, this is handled by *example_spider_data()*:

```
from pyrolite.util.synthetic import example_spider_data

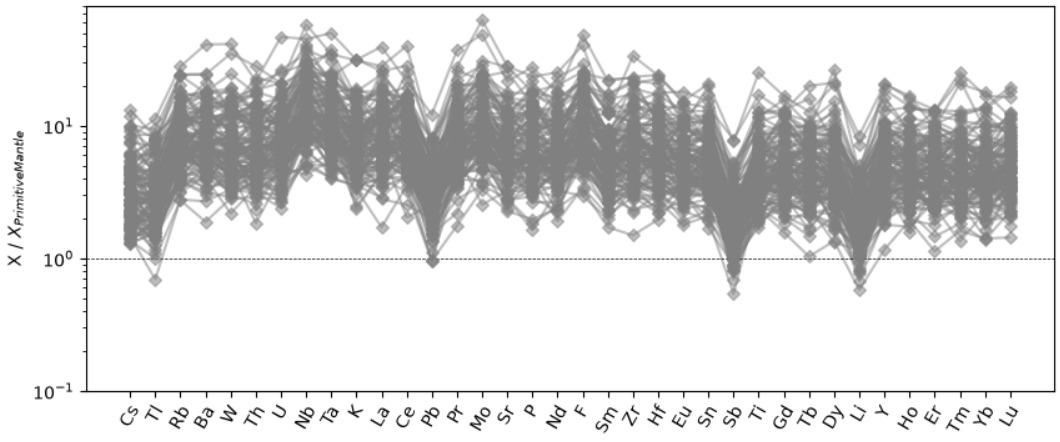
normdf = example_spider_data(start="EMORB_SM89", norm_to="PM_PON")
```

See also

Normalisation

Basic spider plots are straightforward to produce:

```
ax = normdf.pyroplot.spider(color="0.5", alpha=0.5, unity_line=True, figsize=(10, 4))
ax.set_ylabel("X / $X_{Primitive Mantle}$")
plt.show()
```

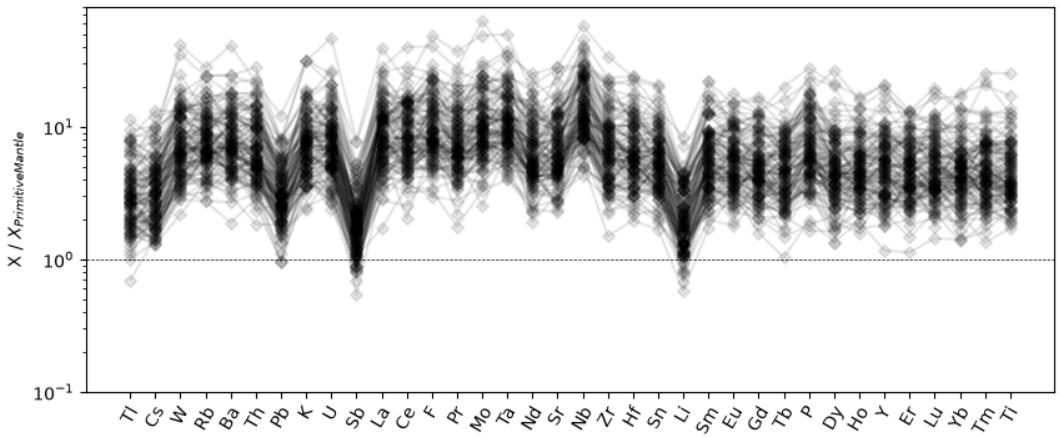


Index Ordering

The default ordering here follows that of the dataframe columns, but we typically want to reorder these based on some physical ordering. A `index_order` keyword argument can be used to supply a function which will reorder the elements before plotting. Here we order the elements by relative incompatibility (using *pyrolite.geochem.ind.by_incompatibility()* behind the scenes):

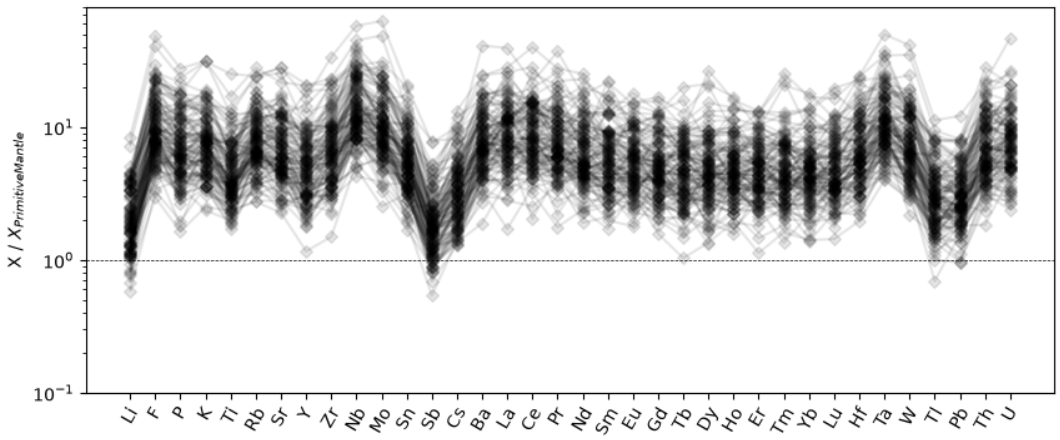
```
from pyrolite.geochem.ind import by_incompatibility

ax = normdf.pyroplot.spider(
    color="k",
    alpha=0.1,
    unity_line=True,
    index_order="incompatibility",
    figsize=(10, 4),
)
ax.set_ylabel("X / $X_{Primitive Mantle}$")
plt.show()
```



Similarly, you can also rearrange elements to be in order of atomic number:

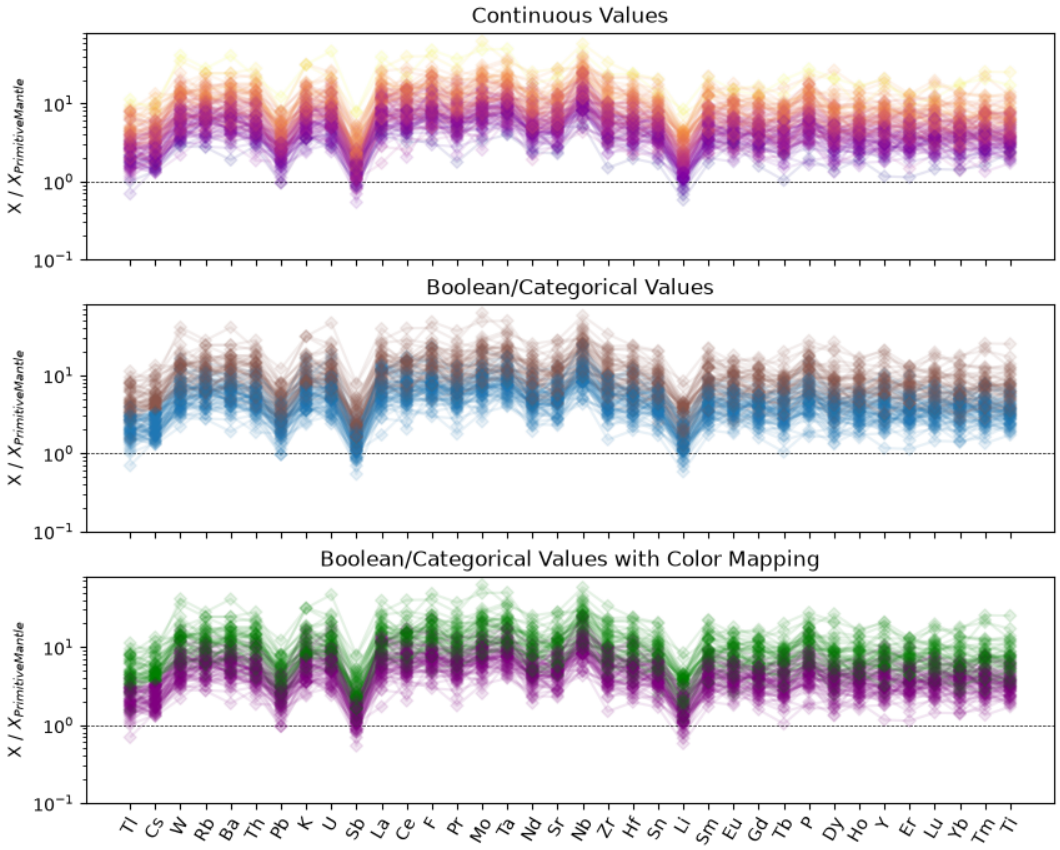
```
ax = normdf.pyroplot.spider(
    color="k",
    alpha=0.1,
    unity_line=True,
    index_order="number",
    figsize=(10, 4),
)
ax.set_ylabel("X / $X_{Primitive Mantle}$")
plt.show()
```



Color Mapping

We can also specify either continuous or categorical values to use for the colors, and even map categorical values to specific colors where useful:

```
fig, ax = plt.subplots(3, 1, sharex=True, sharey=True, figsize=(10, 8))
ax[0].set_title("Continuous Values")
normdf.pyroplot.spider(
    ax=ax[0],
    unity_line=True,
    index_order="incompatibility",
    cmap="plasma",
    alpha=0.1,
    color=np.log(normdf["Li"]),  # a range of continous values
)
ax[1].set_title("Boolean/Categorical Values")
normdf.pyroplot.spider(
    ax=ax[1],
    alpha=0.1,
    unity_line=True,
    index_order="incompatibility",
    color=normdf["Cs"] > 3.5,  # a boolean/categorical set of values
)
ax[2].set_title("Boolean/Categorical Values with Color Mapping")
normdf.pyroplot.spider(
    ax=ax[2],
    alpha=0.1,
    unity_line=True,
    index_order="incompatibility",
    color=normdf["Cs"] > 3.5,  # a boolean/categorical set of values
    color_mappings={  # mapping the boolean values to specific colors
        "color": {True: "green", False: "purple"}
    },
)
[a.set_ylabel("X / $X_{Primitive Mantle}$") for a in ax]
plt.show()
```



Legend Proxies for Spiderplots

While it's relatively straightforward to style spider plots as you wish, for the moment can be a bit more involved to create a legend for these styles. Where you're creating styles based on a set of categories or labels, a few of pyrolite's utility functions might come in handy. Below we'll go through such an example, after creating a few labels (here based on a binning of the Cs abundance):

```
labels = pd.cut(
    np.log(normdf["Cs"]), bins=4, labels=["Low", "Mid. Low", "Mid High", "High"]
)
pd.unique(labels)
```

```
['Mid. Low', 'Mid High', 'Low', 'High']
Categories (4, str): ['Low' < 'Mid. Low' < 'Mid High' < 'High']
```

Below we'll use *process_color()* and *proxy_line()* to construct a set of legend proxies. Note that we need to pass the same configuration to both *spider()* and *process_color()* in order to get the same results, and that the order of labels in the legend will depend on which labels appear first in your dataframe or series (and hence the ordering of the unique values which are returned).

```
from pyrolite.plot.color import process_color
```

(continues on next page)

(continued from previous page)

```
from pyrolite.util.plot.legend import proxy_line

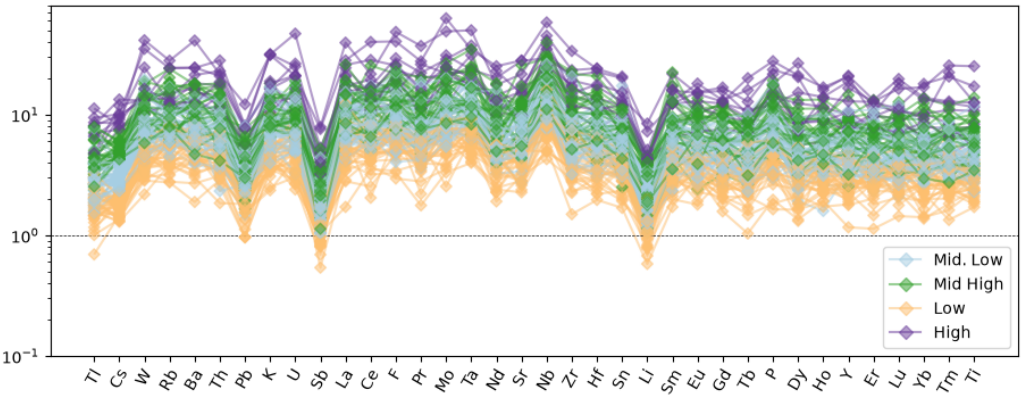
ax = normdf.pyroplot.spider(
    unity_line=True,
    index_order="incompatibility",
    color=labels,  # a categorical set of values
    cmap="Paired",
    alpha=0.5,
    figsize=(11, 4),
)

legend_labels = pd.unique(labels)  # process_color uses this behind the scenes

proxy_colors = process_color(color=legend_labels, cmap="Paired", alpha=0.5)["c"]

legend_proxies = [proxy_line(color=c, marker="D") for c in proxy_colors]

ax.legend(legend_proxies, legend_labels)
plt.show()
```



If the specific order of the labels in your legend is important or you only want to include some of the legend entries for some reason, you could use a dictionary to store the key-value pairs and remap the order of the legend entries manually:

```
proxies = {
    label: proxy_line(color=c, marker="D")
    for label, c in zip(legend_labels, proxy_colors)
}

ordered_labels = ["High", "Mid High", "Mid. Low", "Low"]

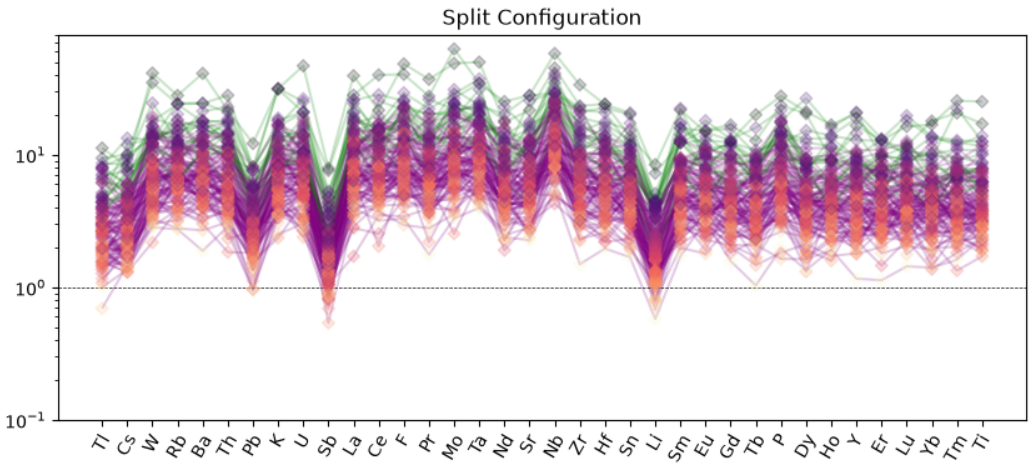
ax.legend([proxies[l] for l in ordered_labels], ordered_labels)

plt.show()
```

Split Configuration

If you have potential conflicts between desired configurations for the lines and markers of your plots, you can explictly separate the configuration using the `scatter_kw` and `line_kw` keyword arguments:

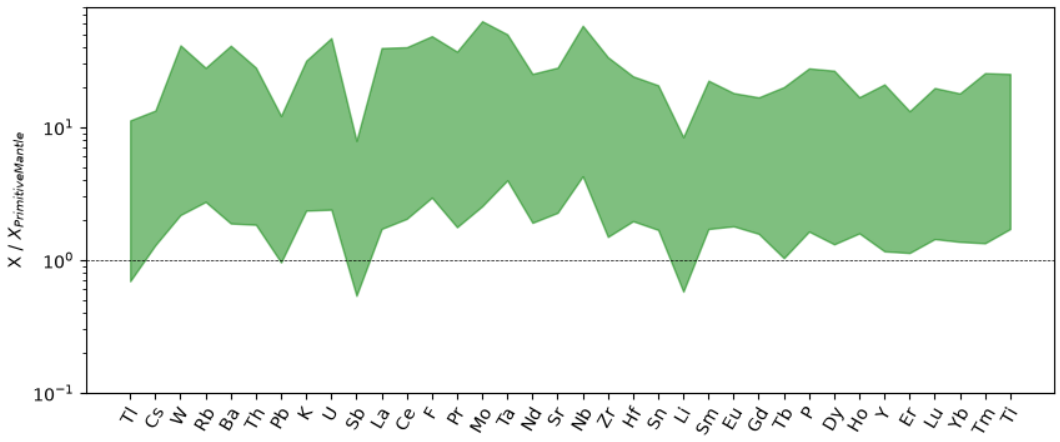
```
fig, ax = plt.subplots(1, 1, sharex=True, sharey=True, figsize=(10, 4))
ax.set_title("Split Configuration")
normdf.pyroplot.spider(
    ax=ax,
    unity_line=True,
    index_order="incompatibility",
    scatter_kw=dict(cmap="magma_r", color=np.log(normdf["Li"])),
    line_kw=dict(
        color=normdf["Cs"] > 5,
        color_mappings={"color": {True: "green", False: "purple"}},
    ),
    alpha=0.2,  # common alpha config between lines and markers
    s=25,  # argument for scatter which won't be passed to lines
)
plt.show()
```



Filled Ranges

The spiderplot can be extended to provide visualisations of ranges and density via the various modes. We could now plot the range of compositions as a filled range:

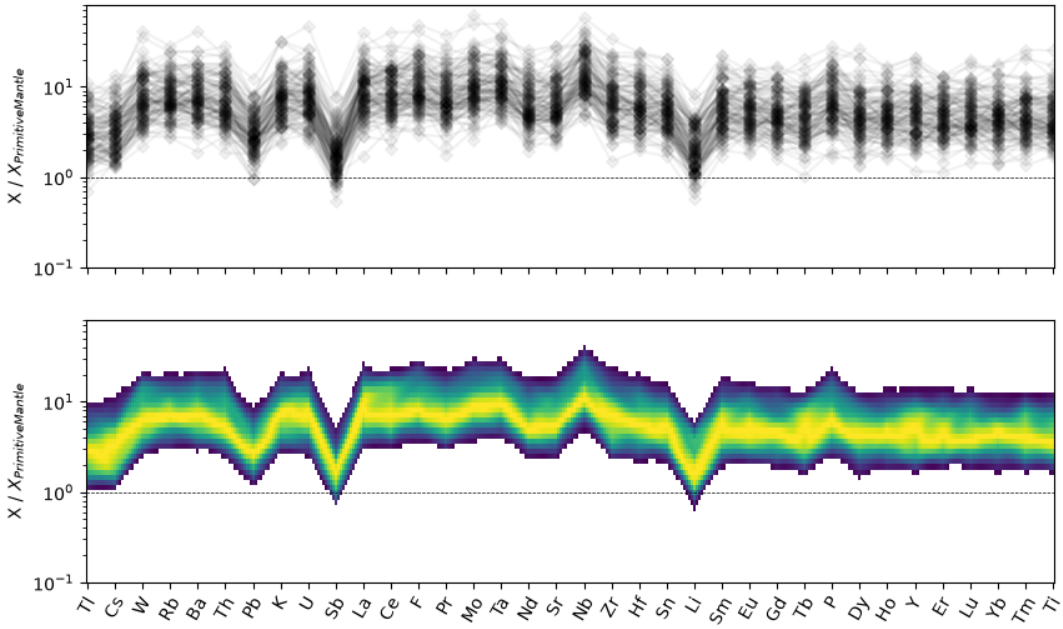
```
ax = normdf.pyroplot.spider(
    mode="fill",
    color="green",
    alpha=0.5,
    unity_line=True,
    index_order="incompatibility",
    figsize=(10, 4),
)
ax.set_ylabel("X / $X_{Primitive Mantle}$")
plt.show()
```



Spider Density Plots

Alternatively, we can plot a conditional density spider plot:

```
fig, ax = plt.subplots(2, 1, sharex=True, sharey=True, figsize=(10, 6))
normdf.pyroplot.spider(
    ax=ax[0], color="k", alpha=0.05, unity_line=True, index_order=by_incompatibility
)
normdf.pyroplot.spider(
    ax=ax[1],
    mode="binkde",
    vmin=0.05,  # 95th percentile,
    resolution=10,
    unity_line=True,
    index_order="incompatibility",
)
[a.set_ylabel("X / $X_{Primitive Mantle}$") for a in ax]
plt.show()
```



We can now assemble a more complete comparison of some of the conditional density modes for spider plots:

```
modes = [
    ("plot", "plot", [], dict(color="k", alpha=0.01)),
    ("fill", "fill", [], dict(color="k", alpha=0.5)),
    ("binkde", "binkde", [], dict(resolution=5)),
    (
        "binkde",
        "binkde contours specified",
        [],
        dict(contours=[0.95], resolution=5),  # 95th percentile contour
    ),
    ("histogram", "histogram", [], dict(resolution=5, bins=30)),
]
```

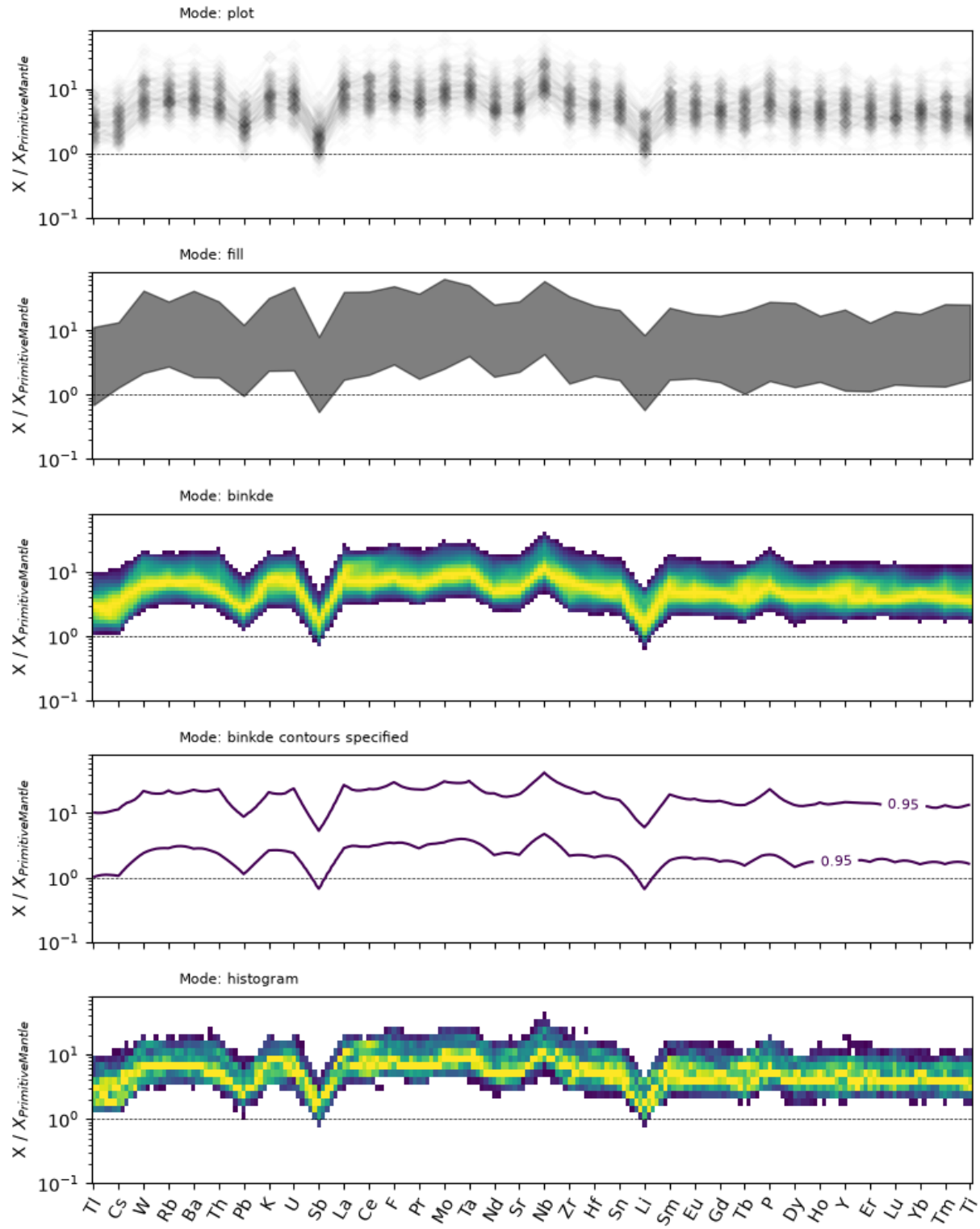
```
down, across = len(modes), 1
fig, ax = plt.subplots(
    down, across, sharey=True, sharex=True, figsize=(across * 8, 2 * down)
)
[a.set_ylabel("X / $X_{Primitive Mantle}$") for a in ax]
for a, (m, name, args, kwargs) in zip(ax, modes):
    a.annotate(  # label the axes rows
        f"Mode: {name}",
        xy=(0.1, 1.05),
        xycoords=a.transAxes,
        fontsize=8,
        ha="left",
        va="bottom",
    )
```

(continues on next page)

(continued from previous page)

```
ax = ax.flat
for mix, (m, name, args, kwargs) in enumerate(modes):
    normdf.pyroplot.spider(
        mode=m,
        ax=ax[mix],
        vmin=0.05,  # minimum percentile
        fontsize=8,
        unity_line=True,
        index_order="incompatibility",
        *args,
        **kwargs,
    )

plt.tight_layout()
```



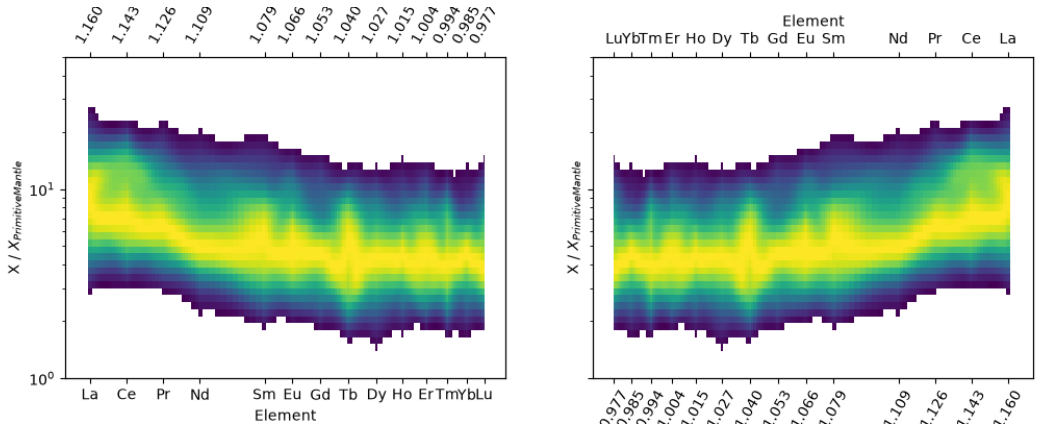

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/plot/spider.py:208:␣
↪UserWarning: No data for colormapping provided via 'c'. Parameters 'vmin' will be␣
↪ignored
  ax.scatter(
```

REE Density Plots

Note that this can also be used for REE-indexed plots, in both configurations. Here we first specify a set of common keyword-argument configurations and use them for both plots:

```
REE_config = dict(unity_line=True, mode="binkde", vmin=0.05, resolution=10)

fig, ax = plt.subplots(1, 2, sharey=True, figsize=(12, 4))
normdf.pyroplot.REE(ax=ax[0], **REE_config)
normdf.pyroplot.REE(ax=ax[1], index="radii", **REE_config)
[a.set_ylabel("X / $X_{Primitive Mantle}$") for a in ax]
plt.show()
```



See also

Heatscatter Plots, Density Diagrams

Total running time of the script: (0 minutes 2.656 seconds)

4.5 Geochemistry Examples

4.5.1 Mineral Database

pyrolite includes a limited mineral database which is useful for looking up endmember compositions.

```
import pandas as pd

from pyrolite.mineral.mindb import (
    get_mineral,
    get_mineral_group,
    list_formulae,
    list_groups,
    list_minerals,
)

pd.set_option("display.precision", 3)  # smaller outputs
```

From the database, you can get the list of its contents using a few utility functions:

```
list_groups()
```

```
['olivine', 'pyroxene', 'spinel', 'feldspar', 'amphibole', 'epidote', 'garnet', 'mica']
```

```
list_minerals()
```

```
['forsterite', 'fayalite', 'tephroite', 'liebenbergite', 'enstatite', 'ferrosilite',
↪'diopside', 'hedenbergite', 'johannsenite', 'esseneite', 'jadeite', 'aegirine',
↪'namansilite', 'kosmochlor', 'spodumene', 'spinel', 'magnesioferrite',
↪'magnesiochromite', 'magnetite', 'hercynite', 'chromite', 'albite', 'microcline',
↪'anorthite', 'anthopyllite', 'gedrite', 'tremolite', 'magnesiohornblende',
↪'ferrohornblende', 'tschermakite', 'ferrotschermakite', 'edenite', 'ferroedenite',
↪'pargasite', 'ferropargasite', 'winchite', 'barroisite', 'richterite', 'ferrorichterite
↪', 'katophorite', 'ferrokatophorite', 'taramite', 'glaucophane', 'eckermanite',
↪'ferroeckermanite', 'magnesioarfvedsonite', 'arvedsonite', 'magnesiohastingsite',
↪'hastingsite', 'magnesioreibeckite', 'riebeckite', 'kaersutite', 'ferrokaersutite',
↪'clinozoisite', 'epidote', 'piemontite', 'allanite', 'pyrope', 'almandine',
↪'spessartine', 'grossular', 'andradite', 'uvarovite', 'majorite', 'morimotoite',
↪'muscovite', 'paragonite', 'celadonite', 'ferroceladonite', 'manganiceladonite',
↪'aluminoceladonite', 'ferroaluminoceladonite', 'chromoceladonite', 'margarite',
↪'phlogopite', 'annite', 'siderophyllite', 'eastonite', 'clintonite', 'phengite',
↪'polylithionite', 'trilithionite', 'chromphyllite']
```

```
list_formulae()
```

```
['Mg2SiO4', 'Fe2SiO4', 'Mn2SiO4', 'Ni1.5Mg0.5SiO4', 'Mg2Si2O6', 'Fe2Si2O6', 'CaMgSi2O6',
↪'CaFeSi2O6', 'CaMnSi2O6', 'CaAlFe{3+}SiO6', 'NaAlSi2O6', 'NaFe{3+}Si2O6', 'NaMn{3+}
↪Si2O6', 'NaCrSi2O6', 'LiAlSi2O6', 'MgAl2O4', 'MgFe{3+}2O4', 'MgCr{3+}2O4', 'Fe{2+}Fe
↪{3+}2O4', 'Fe{2+}Al2O4', 'Fe{2+}Cr{3+}2O4', 'NaAlSi3O8', 'KAlSi3O8', 'CaAl2Si2O8',
↪'(Mg2)(Mg5)(Si8)O22(OH)2', '(Mg2)(Mg3Al2)(Si6Al2)O22(OH)2', '(Ca2)(Mg5)(Si8)O22(OH)2',
↪'(Ca2)(Mg4Al)(Si7Al)O22(OH)2', '(Ca2)(Fe4Al)(Si7Al)O22(OH)2',
↪'(Ca2)(Mg3Al2)(Si6Al2)O22(OH)2', '(Ca2)(Fe3Al2)(Si6Al2)O22(OH)2',
↪'Na(Ca2)(Mg5)(Si7Al)O22(OH)2', 'Na(Ca2)(Fe5)(Si7Al)O22(OH)2',
↪'Na(Ca2)(Mg4Al)(Si6Al2)O22(OH)2', 'Na(Ca2)(Fe4Al)(Si6Al2)O22(OH)2',
↪'(NaCa)(Mg4Al)(Si8)O22(OH)2', '(NaCa)(Mg3Al2)(Si7Al)O22(OH)2',
↪'Na(NaCa)(Mg5)(Si8)O22(OH)2', 'Na(NaCa)(Fe5)(Si8)O22(OH)2',
↪'Na(NaCa)(Mg4Al)(Si7Al)O22(OH)2', 'Na(NaCa)(Fe4Al)(Si7Al)O22(OH)2',
↪'Na(NaCa)(Mg3Al2)(Si6Al2)O22(OH)2', '(Na2)(Mg3Al2)(Si8)O22(OH)2',
↪'Na(Na2)(Mg4Al)(Si8)O22(OH)2', 'Na(Na2)(Fe4Al)(Si8)O22(OH)2', 'Na(Na2)(Mg4Fe{3+}
↪)(Si8)O22(OH)2', 'Na(Na2)(Fe{2+}4Fe{3+})(Si8)O22(OH)2', 'Na(Ca2)(Mg4Fe{3+}
↪)(Si6Al2)O22(OH)2', 'Na(Ca2)(Fe{2+}4Fe{3+})(Si6Al2)O22(OH)2', '(Na2)(Mg3Fe{3+}
↪2)(Si8)O22(OH)2', '(Na2)(Fe3Fe{3+}2)(Si8)O22(OH)2', 'Na(Ca2)(Mg4Ti)(Si6Al2)O22(OH)2',
↪'Na(Ca2)(Fe4Ti)(Si6Al2)O22(OH)2', 'Ca2Al3(Si2O7)(SiO4)O(OH)', 'Ca2Al2Fe{3+}
↪(Si2O7)(SiO4)O(OH)', 'Ca2Al2Mn{3+}(Si2O7)(SiO4)O(OH)', 'CaCe{3+}Al2Fe{2+}
↪(Si2O7)(SiO4)O(OH)', 'Mg3Al2(SiO4)3', 'Fe{2+}3Al2(SiO4)3', 'Mn3Al2(SiO4)3',
↪'Ca3Al2(SiO4)3', 'Ca3Fe{3+}2(SiO4)3', 'Ca3Cr2(SiO4)3', 'Mg3(MgSi)(SiO4)3', 'Ca3(TiFe
↪{2+})(SiO4)3', 'K2(Al4)(Si6Al2)O20(OH)4', 'Na2(Al4)(Si6Al2)O20(OH)4', 'K2(Fe{3+}
↪2Mg2)(Si8)O20(OH)4', 'K2(Fe{3+}2Fe{2+}2)(Si8)O20(OH)4', 'K2(Mn{3+}2Mg2)(Si8)O20(OH)4',
↪'K2(Mg2Al2)(Si8)O20(OH)4', 'K2(Fe{2+}2Al2)(Si8)O20(OH)4', 'K2(Mg2Cr{3+}2)(Si8)O20(OH)4
↪', 'Ca2(Al4)(Si4Al4)O20(OH)4', 'K2(Mg6)(Si6Al2)O20(OH)4', 'K2(Fe{2+}6)(Si6Al2)O20(OH)4
↪', 'K2(Fe{2+}4)(Si4Al6)O20(OH)4', 'K2(Mg4)(Si4Al6)O20(OH)4',
```

(continues on next page)

(continued from previous page)

```
↪'Ca2(Mg4Al2)(Si2Al6)O20(OH)4', 'K2(Al3Mg)(Si7Al)O20(OH)4', 'K2(Al2Li2)(Si8)O20(OH)4',
↪'K2(Al3Li3)(Si6Al2)O20(OH)4', 'K2(Cr{3+}4)(Si6Al2)O20(OH)4']
```

You can also directly get the composition of specific minerals by name:

```
get_mineral("forsterite")
```

```
name       forsterite
group         olivine
formula       Mg2SiO4
Mg              0.346
Si                0.2
O               0.455
Fe                0.0
Mn                0.0
Ni                0.0
Ca                0.0
Al                0.0
Fe{3+}            0.0
Na                0.0
Mn{3+}            0.0
Cr                0.0
Li                0.0
Cr{3+}            0.0
Fe{2+}            0.0
K                 0.0
H                 0.0
Ti                0.0
Ce{3+}            0.0
Name: 0, dtype: object
```

If you want to get compositions for all minerals within a specific group, you can use *get_mineral_group()*:

```
get_mineral_group("olivine")
```

Total running time of the script: (0 minutes 0.014 seconds)

4.5.2 Element-Oxide Transformation

One of pyrolite's strengths is converting mixed elemental and oxide data to a new form. The simplest way to perform this is by using the *convert_chemistry()* function. Note that by default pyrolite assumes that data are in the same units.

```
import pandas as pd

pd.set_option("display.precision", 3)  # smaller outputs
```

Here we create some synthetic data to work with, which has some variables in Wt% and some in ppm. Notably some elements are present in more than one column (Ca, Na):

```
from pyrolite.util.synthetic import normal_frame
```

(continues on next page)

(continued from previous page)

```
df = normal_frame(columns=["MgO", "SiO2", "FeO", "CaO", "Na2O", "Te", "K", "Na"]) * 100
df.pyrochem.elements *= 100  # elements in ppm
```

```
df.head(2)
```

As the units are heterogeneous, we'll need to convert the data frame to a single set of units (here we use Wt%):

```
df.pyrochem.elements = df.pyrochem.elements.pyrochem.scale("ppm", "wt%")  # ppm to wt%
```

We can transform this chemical data to a new set of compositional variables. Here we i) convert CaO to Ca, ii) aggregate Na2O and Na to Na and iii) calculate mass ratios for Na/Te and MgO/SiO2. Note that you can also use this function to calculate mass ratios:

```
df.pyrochem.convert_chemistry(
    to=["MgO", "SiO2", "FeO", "Ca", "Te", "Na", "Na/Te", "MgO/SiO2"]
).head(2)
```

You can also specify molar ratios for iron redox, which will result in multiple iron species within the single dataframe:

```
df.pyrochem.convert_chemistry(to=[{"FeO": 0.9, "Fe2O3": 0.1}]).head(2)
```

Total running time of the script: (0 minutes 0.075 seconds)

4.5.3 Normalization

A selection of reference compositions are included in pyrolite, and can be easily accessed with *pyrolite.geochem.norm.get_reference_composition()* (see the list at the bottom of the page for a complete list):

```
import matplotlib.pyplot as plt

from pyrolite.geochem.norm import all_reference_compositions, get_reference_composition
```

```
chondrite = get_reference_composition("Chondrite_PON")
```

To use the compositions with a specific set of units, you can change them with *set_units()*:

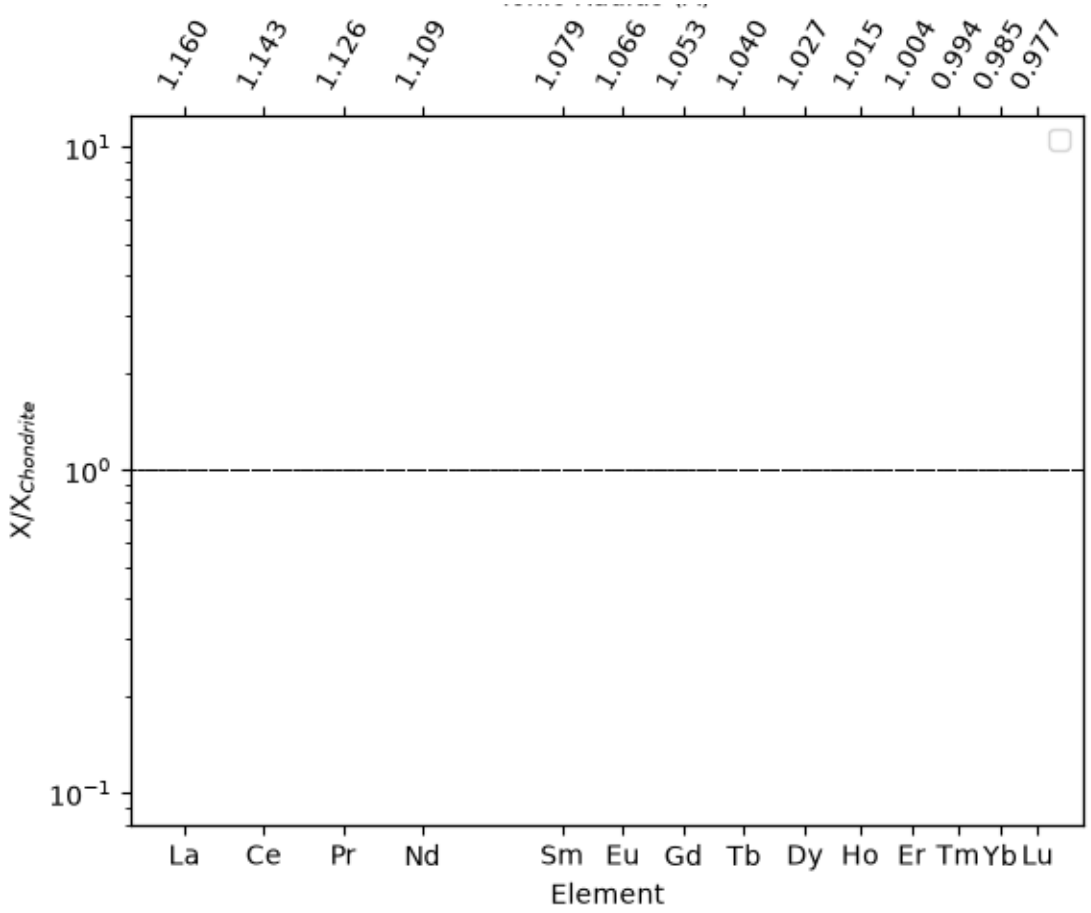
```
CI = chondrite.set_units("ppm")
```

The *normalize_to()* method can be used to normalise DataFrames to a given reference (e.g. for spiderplots):

```
fig, ax = plt.subplots(1)

for name, ref in list(all_reference_compositions().items())[::2]:
    if name != "Chondrite_PON":
        ref.set_units("ppm")
        df = ref.comp.pyrochem.REE.pyrochem.normalize_to(CI, units="ppm")
        df.pyroplot.REE(unity_line=True, ax=ax, label=name)

ax.set_ylabel("X/X$_{Chondrite}$")
ax.legend()
plt.show()
```




```
/home/docs/checkouts/pyrolite/checkouts/develop/docs/source/gallery/examples/geochem/
↪normalization.py:34: UserWarning: No artists with labels found to put in legend.  Note␣
↪that artists whose label start with an underscore are ignored when legend() is called␣
↪with no argument.
  ax.legend()
```

See also

Examples:
lambdas: Parameterising REE Profiles, REE Radii Plot

Currently available models include:

Total running time of the script: (0 minutes 0.462 seconds)

4.5.4 Geochemical Indexes and Selectors

```
import pandas as pd

pd.set_option("display.precision", 3)  # smaller outputs
```

```
from pyrolite.util.synthetic import normal_frame

df = normal_frame(
    columns=[
        "CaO",
        "MgO",
        "SiO2",
        "FeO",
        "Mn",
        "Ti",
        "La",
        "Lu",
        "Y",
        "Mg/Fe",
        "87Sr/86Sr",
        "Ar40/Ar36",
    ]
)
```

```
df.head(2).pyrochem.oxides
```

```
df.head(2).pyrochem.elements
```

```
df.head(2).pyrochem.REE
```

```
df.head(2).pyrochem.REY
```

```
df.head(2).pyrochem.compositional
```

```
df.head(2).pyrochem.isotope_ratios
```

```
df.pyrochem.list_oxides
```

```
['CaO', 'MgO', 'SiO2', 'FeO']
```

```
df.pyrochem.list_elements
```

```
['Mn', 'Ti', 'La', 'Lu', 'Y']
```

```
df.pyrochem.list_REE
```

```
['La', 'Lu']
```

```
df.pyrochem.list_compositional
```

```
['CaO', 'MgO', 'SiO2', 'FeO', 'Mn', 'Ti', 'La', 'Lu', 'Y']
```

```
df.pyrochem.list_isotope_ratios
```

```
['87Sr/86Sr', 'Ar40/Ar36']
```

All elements (up to U):

```
from pyrolite.geochem.ind import REE, REY, common_elements, common_oxides

common_elements()  # string return
```

```
['H', 'He', 'Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne', 'Na', 'Mg', 'Al', 'Si', 'P', 'S',
↪'Cl', 'Ar', 'K', 'Ca', 'Sc', 'Ti', 'V', 'Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu', 'Zn', 'Ga',
↪ 'Ge', 'As', 'Se', 'Br', 'Kr', 'Rb', 'Sr', 'Y', 'Zr', 'Nb', 'Mo', 'Tc', 'Ru', 'Rh', 'Pd
↪', 'Ag', 'Cd', 'In', 'Sn', 'Sb', 'Te', 'I', 'Xe', 'Cs', 'Ba', 'La', 'Ce', 'Pr', 'Nd',
↪'Pm', 'Sm', 'Eu', 'Gd', 'Tb', 'Dy', 'Ho', 'Er', 'Tm', 'Yb', 'Lu', 'Hf', 'Ta', 'W', 'Re
↪', 'Os', 'Ir', 'Pt', 'Au', 'Hg', 'Tl', 'Pb', 'Bi', 'Po', 'At', 'Rn', 'Fr', 'Ra', 'Ac',
↪'Th', 'Pa', 'U']
```

All elements, returned as a list of *~periodictable.core.Formula*:

```
common_elements(output="formula")  # periodictable.core.Formula return
```

```
[H, He, Li, Be, B, C, N, O, F, Ne, Na, Mg, Al, Si, P, S, Cl, Ar, K, Ca, Sc, Ti, V, Cr,␣
↪Mn, Fe, Co, Ni, Cu, Zn, Ga, Ge, As, Se, Br, Kr, Rb, Sr, Y, Zr, Nb, Mo, Tc, Ru, Rh, Pd,␣
↪Ag, Cd, In, Sn, Sb, Te, I, Xe, Cs, Ba, La, Ce, Pr, Nd, Pm, Sm, Eu, Gd, Tb, Dy, Ho, Er,␣
↪Tm, Yb, Lu, Hf, Ta, W, Re, Os, Ir, Pt, Au, Hg, Tl, Pb, Bi, Po, At, Rn, Fr, Ra, Ac, Th,␣
↪Pa, U]
```

Oxides for elements with positive charges (up to U):

```
common_oxides()
```

```
['PmO', 'Pm2O3', 'Fe2O', 'FeO', 'Fe2O3', 'FeO2', 'Fe2O5', 'FeO3', 'Fe2O7', 'U2O', 'UO',
↪'U2O3', 'UO2', 'U2O5', 'UO3', 'Zn2O', 'ZnO', 'P2O', 'PO', 'P2O3', 'PO2', 'P2O5', 'YbO',
↪ 'Yb2O3', 'NdO', 'Nd2O3', 'NdO2', 'Be2O', 'BeO', 'N2O', 'NO', 'N2O3', 'NO2', 'N2O5',
↪'Zr2O', 'ZrO', 'Zr2O3', 'ZrO2', 'Cl2O', 'ClO', 'Cl2O3', 'ClO2', 'Cl2O5', 'ClO3', 'Cl2O7
↪', 'Tc2O', 'TcO', 'Tc2O3', 'TcO2', 'Tc2O5', 'TcO3', 'Tc2O7', 'S2O', 'SO', 'S2O3', 'SO2
↪', 'S2O5', 'SO3', 'Hf2O', 'HfO', 'Hf2O3', 'HfO2', 'HoO', 'Ho2O3', 'Mo2O', 'MoO', 'Mo2O3
↪', 'MoO2', 'Mo2O5', 'MoO3', 'Mn2O', 'MnO', 'Mn2O3', 'MnO2', 'Mn2O5', 'MnO3', 'Mn2O7',
↪'Li2O', 'Y2O', 'YO', 'Y2O3', 'Si2O', 'SiO', 'Si2O3', 'SiO2', 'Tl2O', 'TlO', 'Tl2O3',
↪'As2O', 'AsO', 'As2O3', 'AsO2', 'As2O5', 'La2O', 'LaO', 'La2O3', 'Pb2O', 'PbO', 'Pb2O3
↪', 'PbO2', 'Ca2O', 'CaO', 'Na2O', 'PoO', 'PoO2', 'Po2O5', 'PoO3', 'Cs2O', 'Ta2O', 'TaO
↪', 'Ta2O3', 'TaO2', 'Ta2O5', 'Os2O', 'OsO', 'Os2O3', 'OsO2', 'Os2O5', 'OsO3', 'Os2O7',
↪'OsO4', 'B2O', 'BO', 'B2O3', 'DyO', 'Dy2O3', 'DyO2', 'SmO', 'Sm2O3', 'RaO', 'LuO',
↪'Lu2O3', 'Ge2O', 'GeO', 'Ge2O3', 'GeO2', 'Pa2O3', 'PaO2', 'Pa2O5', 'Ag2O', 'AgO',
↪'Ag2O3', 'AgO2', 'Sc2O', 'ScO', 'Sc2O3', 'Cu2O', 'CuO', 'Cu2O3', 'CuO2', 'ErO', 'Er2O3
↪', 'Ni2O', 'NiO', 'Ni2O3', 'NiO2', 'Se2O', 'SeO', 'Se2O3', 'SeO2', 'Se2O5', 'SeO3',
↪'Ir2O', 'IrO', 'Ir2O3', 'IrO2', 'Ir2O5', 'IrO3', 'Ir2O7', 'IrO4', 'Ir2O9', 'At2O',
↪'At2O3', 'At2O5', 'At2O7', 'TmO', 'Tm2O3', 'Br2O', 'Br2O3', 'BrO2', 'Br2O5', 'Br2O7',
↪'Bi2O', 'BiO', 'Bi2O3', 'BiO2', 'Bi2O5', 'Nb2O', 'NbO', 'Nb2O3', 'NbO2', 'Nb2O5', 'Pd2O
↪', 'PdO', 'Pd2O3', 'PdO2', 'Pd2O5', 'PdO3', 'Rb2O', 'Ga2O', 'GaO', 'Ga2O3', 'I2O',
↪'I2O3', 'IO2', 'I2O5', 'IO3', 'I2O7', 'Gd2O', 'GdO', 'Gd2O3', 'In2O', 'InO', 'In2O3',
↪'Pt2O', 'PtO', 'Pt2O3', 'PtO2', 'Pt2O5', 'PtO3', 'W2O', 'WO', 'W2O3', 'WO2', 'W2O5',
↪'WO3', 'V2O', 'VO', 'V2O3', 'VO2', 'V2O5', 'Sr2O', 'SrO', 'Cr2O', 'CrO', 'Cr2O3', 'CrO2
```

(continues on next page)

(continued from previous page)

```
↪', 'Cr2O5', 'CrO3', 'Hg2O', 'HgO', 'HgO2', 'Mg2O', 'MgO', 'Th2O', 'ThO', 'Th2O3', 'ThO2
↪', 'Ru2O', 'RuO', 'Ru2O3', 'RuO2', 'Ru2O5', 'RuO3', 'Ru2O7', 'RuO4', 'Sn2O', 'SnO',
↪'Sn2O3', 'SnO2', 'Au2O', 'AuO', 'Au2O3', 'Au2O5', 'K2O', 'Re2O', 'ReO', 'Re2O3', 'ReO2
↪', 'Re2O5', 'ReO3', 'Re2O7', 'Rh2O', 'RhO', 'Rh2O3', 'RhO2', 'Rh2O5', 'RhO3', 'C2O',
↪'CO', 'C2O3', 'CO2', 'Te2O', 'TeO', 'Te2O3', 'TeO2', 'Te2O5', 'TeO3', 'Ac2O3', 'H2O',
↪'Ba2O', 'BaO', 'Tb2O', 'TbO', 'Tb2O3', 'TbO2', 'Sb2O', 'SbO', 'Sb2O3', 'SbO2', 'Sb2O5',
↪ 'Cd2O', 'CdO', 'CeO', 'Ce2O3', 'CeO2', 'RnO', 'RnO3', 'EuO', 'Eu2O3', 'Ti2O', 'TiO',
↪'Ti2O3', 'TiO2', 'PrO', 'Pr2O3', 'PrO2', 'Pr2O5', 'Co2O', 'CoO', 'Co2O3', 'CoO2',
↪'Co2O5', 'Fr2O', 'Al2O', 'AlO', 'Al2O3', 'FeOT', 'Fe2O3T', 'LOI']
```

```
REE()
```

```
['La', 'Ce', 'Pr', 'Nd', 'Sm', 'Eu', 'Gd', 'Tb', 'Dy', 'Ho', 'Er', 'Tm', 'Yb', 'Lu']
```

```
REY()
```

```
['La', 'Ce', 'Pr', 'Nd', 'Sm', 'Eu', 'Gd', 'Tb', 'Dy', 'Y', 'Ho', 'Er', 'Tm', 'Yb', 'Lu']
```

Total running time of the script: (0 minutes 0.137 seconds)

4.5.5 Mineral Endmember Decomposition

A common task when working with mineral chemistry data is to take measured compositions and decompose these into relative proportions of mineral endmember compositions. pyrolite includes some utilities to achieve this and a limited mineral database for looking up endmember compositions.

```
import numpy as np
import pandas as pd

from pyrolite.mineral.mindb import get_mineral
from pyrolite.mineral.normative import endmember_decompose
```

First we'll start with a composition of an unknown olivine:

```
comp = pd.Series({"MgO": 42.06, "SiO2": 39.19, "FeO": 18.75})
```

We can break this down into olivine endmebmers using the `endmember_decompose()` function:

```
ed = endmember_decompose(
    pd.DataFrame(comp).T, endmembers="olivine", order=1, molecular=True
)
ed
```

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:122:␣
↪UserWarning: Non-positive entries found in specified components. Negative values have␣
↪been replaced with NaN. Renormalisation assumes all positive entries.
  warnings.warn(
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:122:␣
↪UserWarning: Non-positive entries found in specified components. Negative values have␣
↪been replaced with NaN. Renormalisation assumes all positive entries.
  warnings.warn(
```

(continues on next page)

(continued from previous page)

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:56: UserWarning:␣
↪Non-positive entries found. Closure operation assumes all positive entries.
  warnings.warn(
```

Equally, if you knew the likely endmembers beforehand, you could specify a list of endmembers:

```
ed = endmember_decompose(
    pd.DataFrame(comp).T, endmembers=["forsterite", "fayalite"], order=1, molecular=True
)
ed
```

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:122:␣
↪UserWarning: Non-positive entries found in specified components. Negative values have␣
↪been replaced with NaN. Renormalisation assumes all positive entries.
  warnings.warn(
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:122:␣
↪UserWarning: Non-positive entries found in specified components. Negative values have␣
↪been replaced with NaN. Renormalisation assumes all positive entries.
  warnings.warn(
```

We can check this by recombining the components with these proportions. We can first lookup the compositions for our endmembers:

```
em = pd.DataFrame([get_mineral("forsterite"), get_mineral("fayalite")])
em.loc[:, ~(em == 0).all(axis=0)]  # columns not full of zeros
```

First we have to convert these element-based compositions to oxide-based compositions:

```
emvalues = (
    em.loc[:, ["Mg", "Si", "Fe"]]
    .pyrochem.to_molecular()
    .fillna(0)
    .pyrochem.convert_chemistry(to=["MgO", "SiO2", "FeO"], molecular=True)
    .fillna(0)
    .pyrocomp.renormalise(scale=1)
)
emvalues
```

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:122:␣
↪UserWarning: Non-positive entries found in specified components. Negative values have␣
↪been replaced with NaN. Renormalisation assumes all positive entries.
  warnings.warn(
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:122:␣
↪UserWarning: Non-positive entries found in specified components. Negative values have␣
↪been replaced with NaN. Renormalisation assumes all positive entries.
  warnings.warn(
```

These can now be used with our endmember proportions to regenerate a composition:

```
recombined = pd.DataFrame(ed.values.flatten() @ emvalues).T.pyrochem.to_weight()
recombined
```

To make sure these compositions are within 0.01 percent:

```
assert np.allclose(recombined.values, comp.values, rtol=10**-4)
```

Total running time of the script: (0 minutes 0.151 seconds)

4.5.6 Ionic Radii

`pyrolite` incldues a few sets of reference tables for ionic radii in aangstroms (Å) from [Shannon1976] and [WhittakerMuntus1970], each with tables indexed by element, ionic charge and coordination. The easiset way to access these is via the *get_ionic_radii()* function. The function can be used to get radii for individual elements:

```
from pyrolite.geochem.ind import REE, get_ionic_radii

Cu_radii = get_ionic_radii("Cu")
print(Cu_radii)
```

```
index
Cu2+IV      0.57
Cu2+IVSQ    0.57
Cu2+V       0.65
Cu2+VI      0.73
Name: ionicradius, dtype: float64
```

Note that this function returned a series of the possible radii, given specific charges and coordinations of the Cu ion. If we completely specify these, we'll get a single number back:

```
Cu2plus6fold_radii = get_ionic_radii("Cu", coordination=6, charge=2)
print(Cu2plus6fold_radii)
```

```
0.73
```

You can also pass lists to the function. For example, if you wanted to get the Shannon ionic radii of Rare Earth Elements (REE) in eight-fold coordination with a valence of +3, you should use the following:

```
shannon_ionic_radii = get_ionic_radii(REE(), coordination=8, charge=3)
print(shannon_ionic_radii)
```

```
[1.16  1.143 1.126 1.109 1.079 1.066 1.053 1.04  1.027 1.015 1.004 0.994
 0.985 0.977]
```

The function defaults to using the Shannon ionic radii consistent with [Pauling1960], but you can adjust to use the set you like with the *pauling* boolean argument (`pauling=False` to use Shannon's 'Crystal Radii') or the *source* argument (`source='Whittaker'` to use the [WhittakerMuntus1970] dataset):

```
shannon_crystal_radii = get_ionic_radii(REE(), coordination=8, charge=3, pauling=False)
whittaker_ionic_radii = get_ionic_radii(
    REE(), coordination=8, charge=3, source="Whittaker"
)
```

We can see what the differences between these look like across the REE:

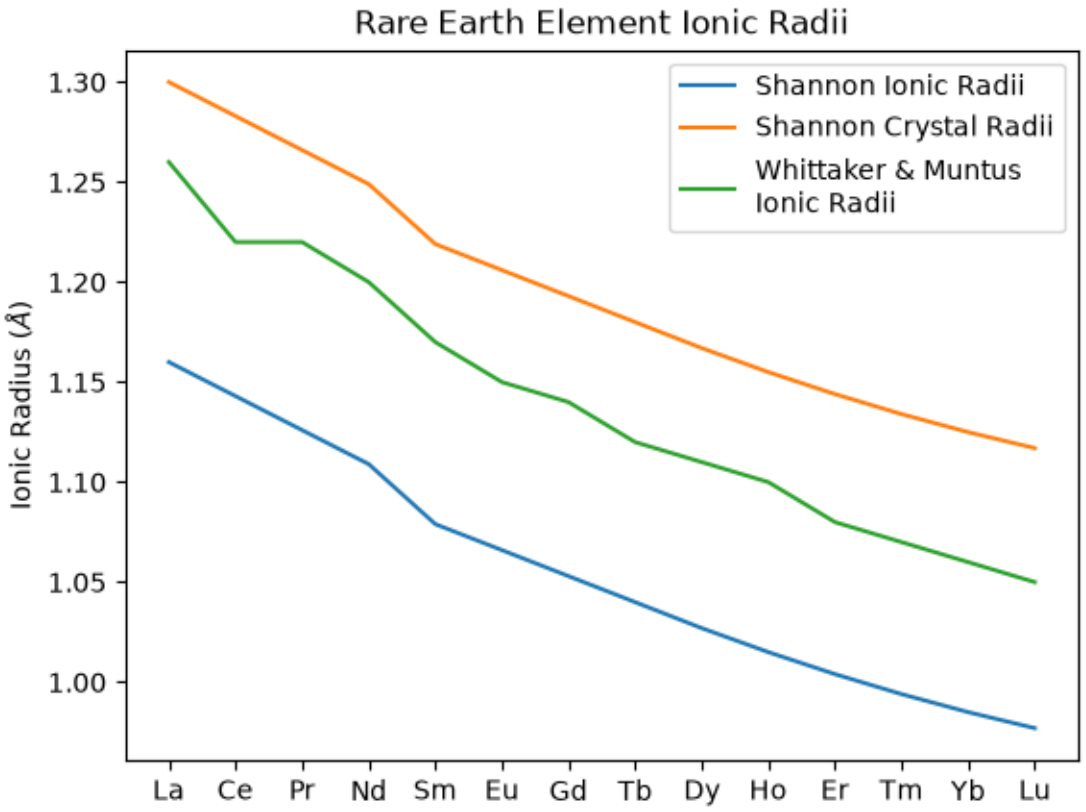
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1)
```

(continues on next page)

(continued from previous page)

```
ax.plot(shannon_ionic_radii, label="Shannon Ionic Radii")
ax.plot(shannon_crystal_radii, label="Shannon Crystal Radii")
ax.plot(whittaker_ionic_radii, label="Whittaker & Muntus\nIonic Radii")
{a: b for (a, b) in zip(REE(), whittaker_ionic_radii)}
ax.set_xticks(range(len(REE())))
ax.set_xticklabels(REE())
ax.set_ylabel(r"Ionic Radius ($\AA$)")
ax.set_title("Rare Earth Element Ionic Radii")
ax.legend()
plt.show()
```



See also

Examples:
lambdas: Parameterising REE Profiles, REE Radii Plot

Functions:
get_ionic_radii(), *pyrolite.geochem.ind.REE()*, *lambda_lnREE()*,

References

Total running time of the script: (0 minutes 0.104 seconds)

4.5.7 Unit Scaling

```
import pandas as pd

pd.set_option("display.precision", 3)  # smaller outputs
```

Here we create an example dataframe to work from, containing some synthetic data in the form of oxides and elements, each with different units (wt% and ppm, respectively).

```
from pyrolite.util.synthetic import normal_frame

df = normal_frame(
    columns=["CaO", "MgO", "SiO2", "FeO", "Ni", "Ti", "La", "Lu"], seed=22
)
df.pyrochem.oxides *= 100  # oxides in wt%
df.pyrochem.elements *= 10000  # elements in ppm
```

In this case, we might want to transform the Ni and Ti into their standard oxide equivalents NiO and TiO2:

```
df.pyrochem.convert_chemistry(to=["NiO", "TiO2"]).head(2)
```

But here because Ni and Ti have units of ppm, the results are a little non-sensical, especially when it's combined with the other oxides:

```
df.pyrochem.convert_chemistry(to=df.pyrochem.list_oxides + ["NiO", "TiO2"]).head(2)
```

There are multiple ways we could convert the units, but here we're going to first convert the elemental ppm data to wt%, then perform our oxide-element conversion. To do this, we'll use the built-in function *scale()*:

```
from pyrolite.util.units import scale

df.pyrochem.elements *= scale("ppm", "wt%")
```

We can see that this then gives us numbers which are a bit more sensible:

```
df.pyrochem.convert_chemistry(to=df.pyrochem.list_oxides + ["NiO", "TiO2"]).head(2)
```

Dealing with Units in Column Names

Often our dataframes will start containing column names which pyrolite doesn't recognize natively by default (work in progress, this is an item on the roadmap). Here we can create an example of that, and go through some key steps for using this data in `pyrolite`:

```
df = normal_frame(
    columns=["CaO", "MgO", "SiO2", "FeO", "Ni", "Ti", "La", "Lu"], seed=22
)
df.pyrochem.oxides *= 100  # oxides in wt%
df.pyrochem.elements *= 10000  # elements in ppm
df = df.rename(
    columns={
        **{c: c + "_wt%" for c in df.pyrochem.oxides},
        **{c: c + "_ppm" for c in df.pyrochem.elements},
    }
)
df.head(2)
```

If you just wanted to rescale some columns, you can get away without renaming your columns, e.g. converting all of the ppm columns to wt%:

```
df.loc[:, [c for c in df.columns if "_ppm" in c]] *= scale("ppm", "wt%")
df.head(2)
```

However, to access the full native capability of pyrolite, we'd need to rename these columns to use things like *convert_chemistry()*:

```
units = {  # keep a copy of the units, we can use these to map back later
    c: c[c.find("_") + 1 :] if "_" in c else None for c in df.columns
}
df = df.rename(
    columns={c: c.replace("_wt%", "").replace("_ppm", "") for c in df.columns}
)
df.head(2)
```

We could then perform our chemistry conversion, rename our columns to include units, and optionally export to e.g. CSV:

```
converted_wt_pct = df.pyrochem.convert_chemistry(
    to=df.pyrochem.list_oxides + ["NiO", "TiO2"]
)
converted_wt_pct.head(2)
```

Here we rename the columns before we export them, just so we know explicitly what the units are:

```
converted_wt_pct = converted_wt_pct.rename(
    columns={c: c + "_wt%" for c in converted_wt_pct.pyrochem.list_oxides}
)
converted_wt_pct.head(2)
```

```
converted_wt_pct.to_csv("converted_wt_pct.csv")
```

Total running time of the script: (0 minutes 0.160 seconds)

4.5.8 CIPW Norm

The CIPW (W. Cross, J. P. Iddings, L. V. Pirsson, and H. S. Washington) Norm was introducted as a standard procedure for the estimation of rock-forming mineral assemblages of igneous rocks from their geochemical compositions [Cross1902] . This estimation process enables the approximate classificaiton of microcrystalline and partially crystalline rocks using a range of mineralogically-based classificaiton systems (e.g. most IUGS classifications), and the generation of normative-mineral modifiers for geochemical classificaiton systems.

A range of updated, modified and adjusted Norms were published in the century following the original publication of the CIPW Norm, largely culminating in Surendra Verma's 2003 paper "A revised CIPW norm" which enumerates an algorithm for the estimation of an anhydrous Standard Igenous Norm (SIN) [Verma2003] . This was subsequently updated with the publication of IgRoCS [Verma2013] . A version of this algorithm has now been implemented in `pyrolite` (*CIPW_norm()*), and an overview of the implementation and the currently available options is given below.

For the purposes of testing, pyrolite includes a file containing the outputs from Verma's SINCLAS/IgRoCS program. Here we can use this file to demonstrate the use of the CIPW Norm and verify that the results should generally be comparable between Verma's original implementation and the `pyrolite` implementation. We import this file and do a little cleaning and registration of geochemical components so we can work with it in the sections to follow:

```
import warnings

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from pyrolite.util.meta import pyrolite_datafolder

# silence a pandas warning
warnings.simplefilter("ignore", category=pd.errors.PerformanceWarning)

df = (
    pd.read_csv(pyrolite_datafolder() / "testing" / "CIPW_Verma_Test.csv")
    .dropna(how="all", axis=1)
    .pyrochem.parse_chem()
)
df.pyrochem.compositional = df.pyrochem.compositional.apply(
    pd.to_numeric, errors="coerce"
).fillna(0)
df[[c for c in df.columns if "NORM" in c]] = df[
    [c for c in df.columns if "NORM" in c]
].apply(pd.to_numeric, errors="coerce")
```

The CIPW Norm can be accessed via *pyrolite.mineral.normative.CIPW_norm()*, and expects a dataframe as input containing major element oxides (in wt%) and can also use a select set of trace elements (in ppm).

```
from pyrolite.mineral.normative import CIPW_norm

NORM = CIPW_norm(df.pyrochem.compositional)
```

We can quickly check that this includes mineralogical data:

```
NORM.columns
```

```
Index(['quartz', 'zircon', 'potassium metasilicate', 'anorthite',
       'sodium metasilicate', 'acmite', 'thenardite', 'albite', 'orthoclase',
```

(continues on next page)

(continued from previous page)

```
       'perovskite', 'nepheline', 'leucite', 'dicalcium silicate',
       'kaliophilite', 'apatite', 'fluroapatite', 'fluorite', 'pyrite',
       'chromite', 'ilmenite', 'calcite', 'corundum', 'rutile', 'magnetite',
       'hematite', 'forsterite', 'fayalite', 'clinoferrosilite',
       'clinoenstatite', 'ferrosilite', 'enstatite', 'wollastonite',
       'cancrinite', 'halite', 'titanite', 'diopside', 'hypersthene',
       'olivine'],
      dtype='str')
```

The function accepts a few keyword arguments, all to do with the iron compositions and related adjustment/corrections:

Fe_correction = "LeMaitre" | "Middlemost"
: For specifying the Fe-correction method/function. Currently includes LeMaitre's correction method [LeMaitre1976] (the default) and Middlemost's TAS-based correction [Middlemost1989] .

Fe_correction_mode = 'volcanic'
: For specificying the Fe-correction mode, for LeMaitre's correction.

adjust_all_Fe = False
: Specifying whether you want to adjust all iron compositions, or only those which are partially specified (i.e. only have a singular value for one of FeO, Fe2O3, FeOT, Fe2O3T).

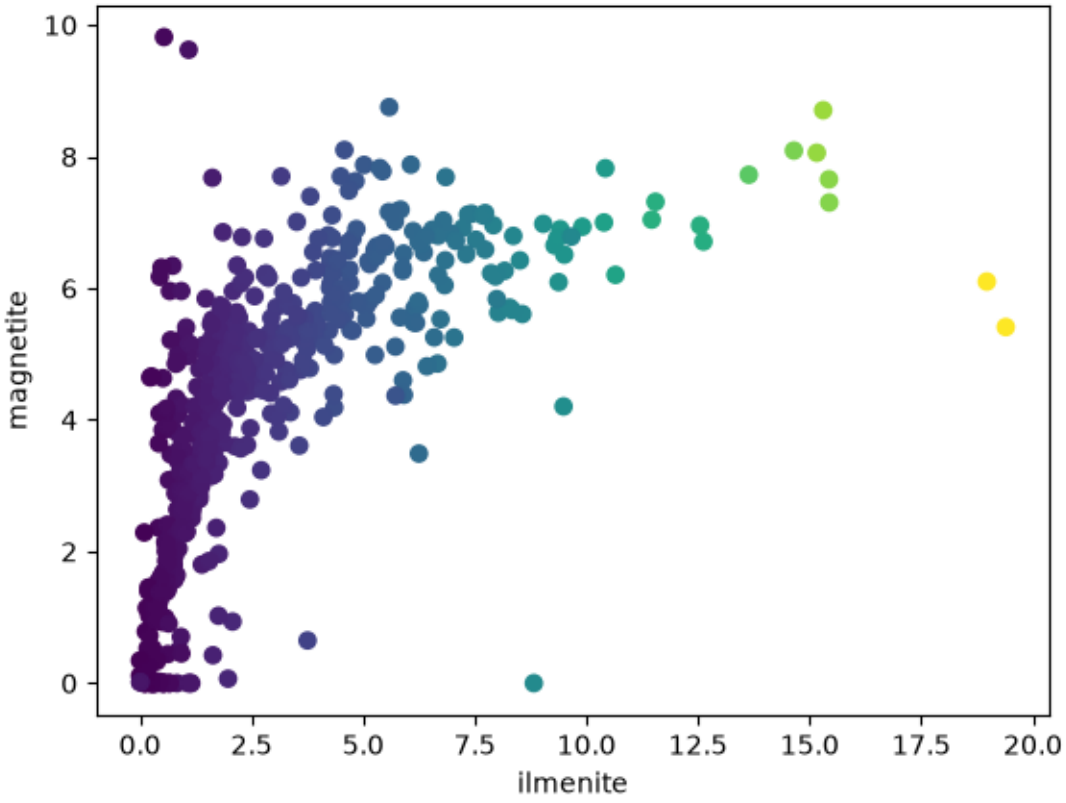
```
NORM = CIPW_norm(df.pyrochem.compositional, Fe_correction="Middlemost")
```

For the purpose of establishing the congruency of our algorithm with Verma's, we'll use `adjust_all_Fe = True` and LeMaitre's correction. Notably, this won't make too much difference to the format of the output, but it will adjust the estimates of normative mineralogy depending on oxidation state.

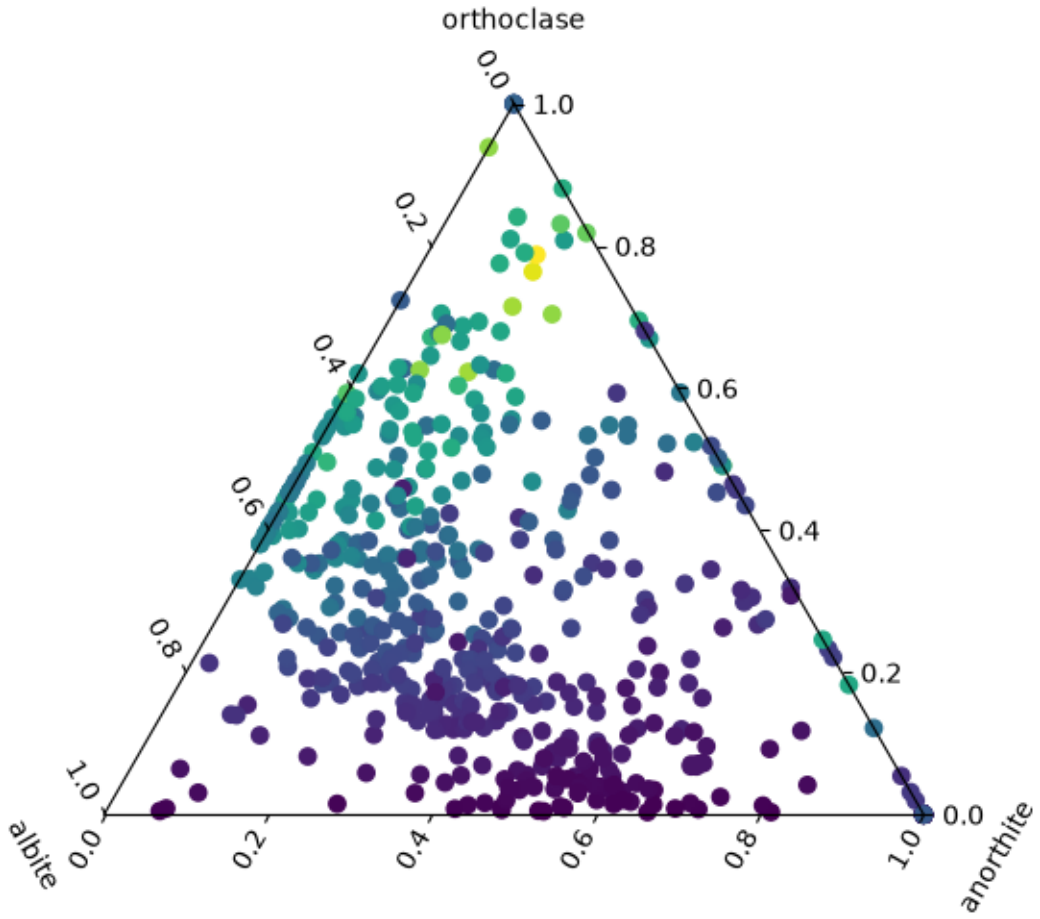
```
NORM = CIPW_norm(
    df.pyrochem.compositional,
    adjust_all_Fe=True,
    Fe_correction="LeMaitre",
    Fe_correction_mode="volcanic",
)
```

Now we have the normative mineralogical outputs, we can have a look to see how these compare to some relevant geochemical inputs:

```
ax = NORM[["ilmenite", "magnetite"]].pyroplot.scatter(clip_on=False, c=df["TiO2"])
plt.show()
```




```
ax = NORM[["orthoclase", "albite", "anorthite"]].pyroplot.scatter(
    clip_on=False, c=df["K2O"]
)
plt.show()
```



Coherency with SINCLAS / IgRoCS

Given we're reproducting an existing algorithm, it's prudent to check how closely the results match for a specific dataset to check whether there might be any numerical or computational errors. Below we go through this exercise for the test dataset we loaded above (which already includes the output of SINCLAS), comparing the original software to the pyrolite implementation.

Note

Currently there are inconsistent results for a small number or samples (deviations colormapped, and inconsistent results shown in red below), likely related to the handling of iron components and their conversion.

The output of SINCLAS has slightly different column naming that that of `pyrolite`, which provides full mineral names in the output dataframe columns. For this reason, we'll need to translate our NORM output columns to the SINCLAS column names. For this we can use the dictionary of minerals used in the CIPW Norm (`NORM_MINERALS`)

```
from pyrolite.mineral.normative import NORM_MINERALS

translation = {
    d["name"]: (d.get("SINCLAS_abbrv", None) or k.upper()) + "_NORM"
    for k, d in NORM_MINERALS.items()
    if (d.get("SINCLAS_abbrv", None) or k.upper()) + "_NORM" in df.columns
}
```

(continues on next page)

(continued from previous page)

```
translation
```

```
{'quartz': 'Q_NORM', 'zircon': 'ZIR_NORM', 'potassium metasilicate': 'KS_NORM',
↪'anorthite': 'AN_NORM', 'sodium metasilicate': 'NS_NORM', 'acmite': 'AC_NORM',
↪'thenardite': 'TH_NORM', 'albite': 'AB_NORM', 'orthoclase': 'OR_NORM', 'perovskite':
↪'PER_NORM', 'nepheline': 'NE_NORM', 'leucite': 'LC_NORM', 'dicalcium silicate': 'CS_
↪NORM', 'kaliophilite': 'KP_NORM', 'apatite': 'AP_NORM', 'fluorite': 'FR_NORM', 'pyrite
↪': 'PYR_NORM', 'chromite': 'CHR_NORM', 'ilmenite': 'IL_NORM', 'calcite': 'CC_NORM',
↪'corundum': 'C_NORM', 'rutile': 'RU_NORM', 'magnetite': 'MT_NORM', 'forsterite': 'FO_
↪NORM', 'fayalite': 'FA_NORM', 'clinoferrosilite': 'DIF_NORM', 'clinoenstatite': 'DIM_
↪NORM', 'ferrosilite': 'HYF_NORM', 'enstatite': 'HYM_NORM', 'wollastonite': 'WO_NORM',
↪'cancrinite': 'CAN_NORM', 'halite': 'HL_NORM', 'titanite': 'SPH_NORM', 'diopside': 'DI_
↪NORM', 'hypersthene': 'HY_NORM', 'olivine': 'OL_NORM'}
```

First we'll collect the minerals which appear in both dataframes, and then iterate through these to check how close the implementations are.

```
minerals = {
    k: v for (k, v) in translation.items() if (df[v] > 0).sum() and (NORM[k] > 0).sum()
}
```

To compare SINCLAS and the `pyrolite` NORM outputs, we'll construct a grid of plots which compare the respective mineralogical norms relative to a 1:1 line, and highlight discrepancies. As we'll do it twice below (once for samples labelled as volanic, and once for everything else), we may as well make a function of it.

After that, let's take a look at the volcanic samples in isolation, which are the the key samples for which the NORM should be applied:

```
from pyrolite.plot.color import process_color


def compare_NORMs(SINCLAS_outputs, NORM_outputs, name=""):
    """
    Create a grid of axes comparing the outputs of SINCLAS and `pyrolite`'s NORM,
    after translating the column names to the appropriate form.
    """
    ncols = 4
    nrows = len(minerals.keys()) // ncols + (1 if len(minerals.keys()) % ncols else 0)

    fig, ax = plt.subplots(nrows, ncols, figsize=(ncols * 2.5, nrows * 2))
    fig.suptitle(
        " - ".join(
            ["Comparing pyrolite's CIPW Norm to SINCLAS/IgRoCS"] + [name]
            if name
            else []
        ),
        fontsize=16,
        y=1.01,
    )
    ax = ax.flat
    for ix, (b, a) in enumerate(minerals.items()):
        ax[ix].set_title("\n".join(b.split()), y=0.9, va="top")
        if a in SINCLAS_outputs.columns and b in NORM_outputs.columns:
```

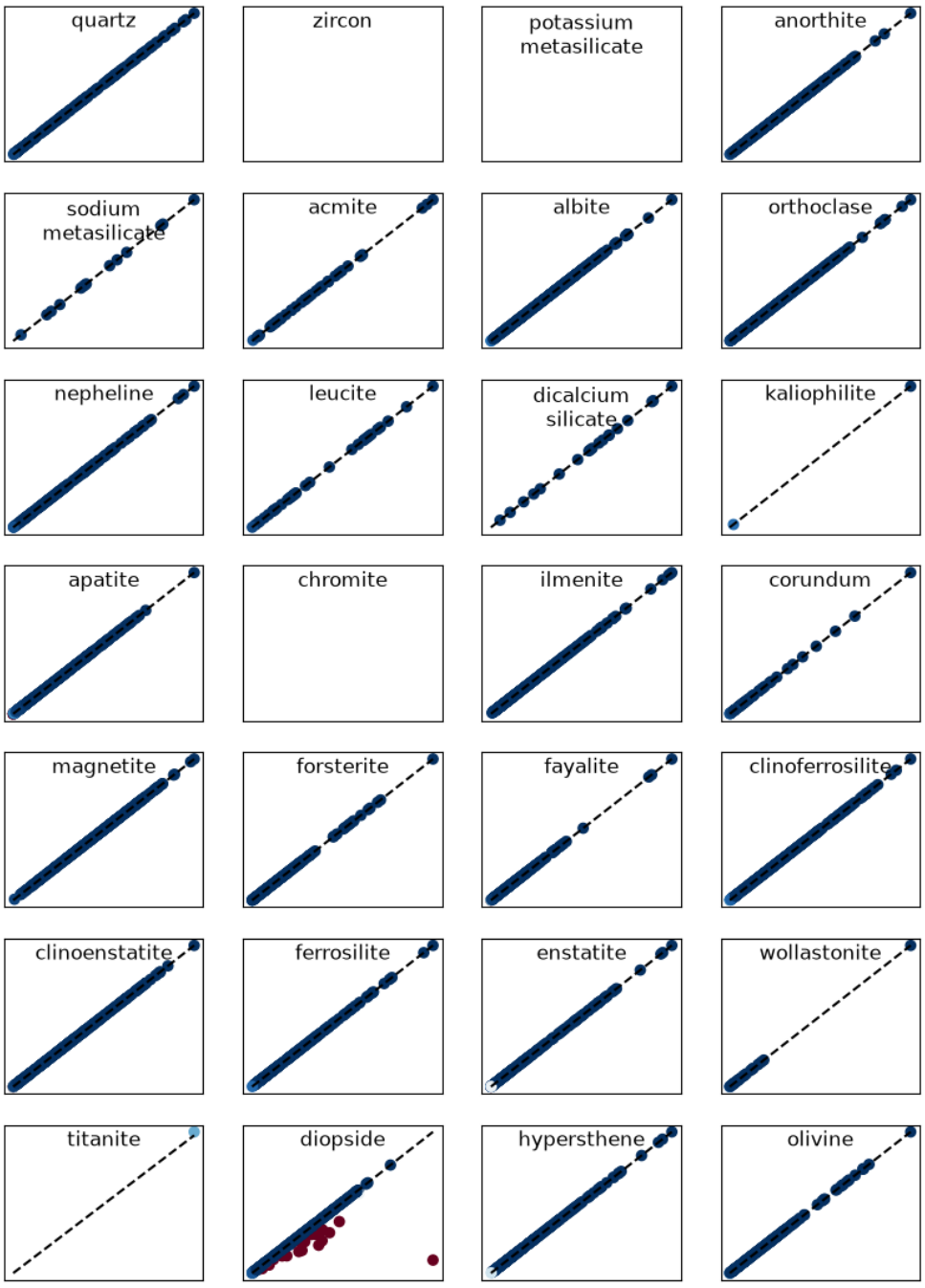
(continues on next page)

(continued from previous page)

```
            # colour by deviation from unity
            c = process_color(
                np.abs((SINCLAS_outputs[a] / NORM_outputs[b]) - 1),
                cmap="RdBu_r",
                norm=plt.Normalize(vmin=0, vmax=0.1),
            )["c"]
            ax[ix].scatter(SINCLAS_outputs[a], NORM_outputs[b], c=c)
        # add a 1:1 line
        ax[ix].plot(
            [0, SINCLAS_outputs[a].max()],
            [0, SINCLAS_outputs[a].max()],
            color="k",
            ls="--",
        )

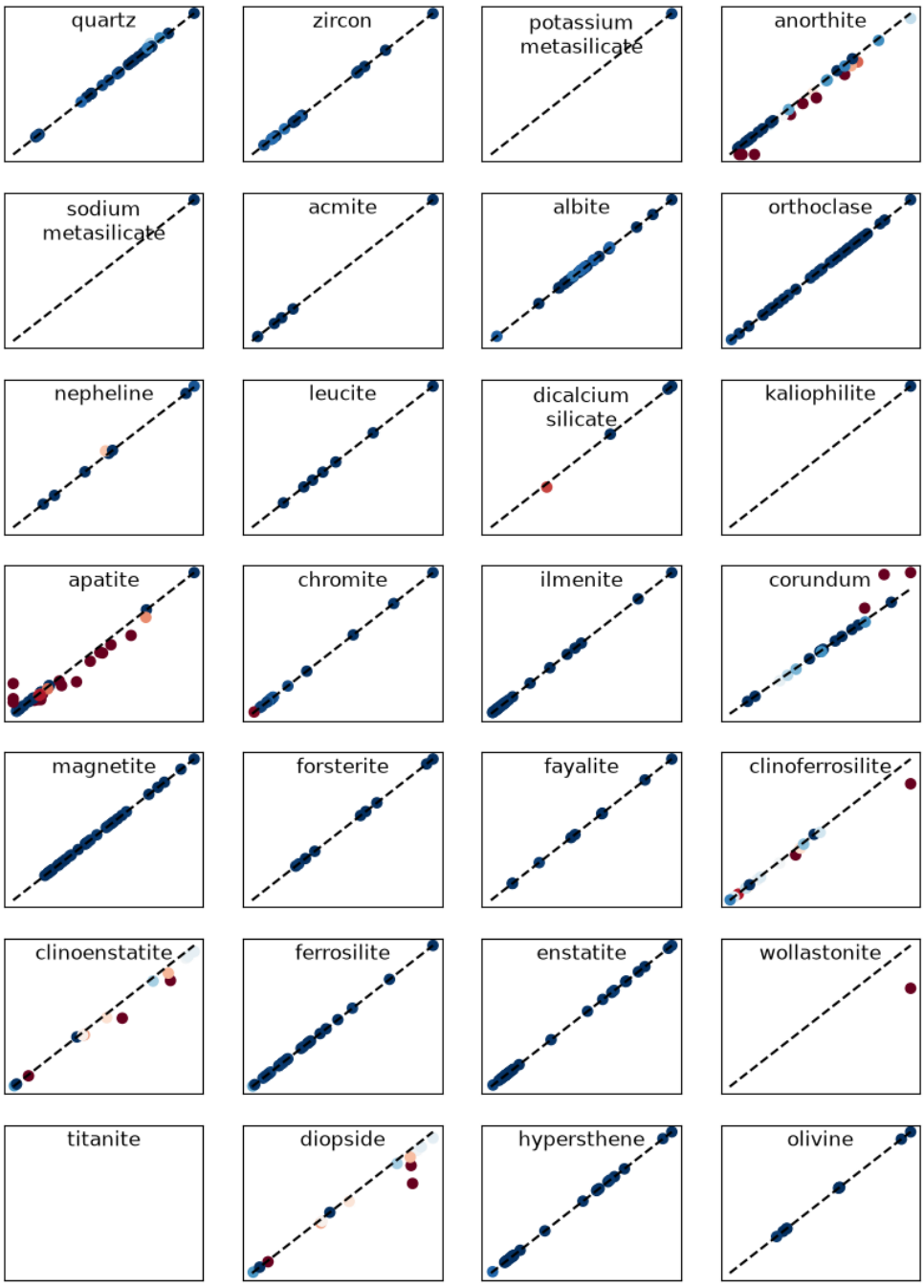
    for a in ax:
        a.set(xticks=[], yticks=[])  # turn off the ticks
        if not a.collections:  # turn off the axis for empty axes
            a.axis("off")
    return fig, ax


volcanic_filter = df.loc[:, "ROCK_TYPE"].str.lower().str.startswith("volc")
fig, ax = compare_NORMs(df.loc[volcanic_filter, :], NORM.loc[volcanic_filter])
```


quartz
zircon
potassium metasilicate
anorthite
sodium metasilicate
acmite
albite
orthoclase
nepheline
leucite
dicalcium silicate
kaliophilite
apatite
chromite
ilmenite
corundum
magnetite
forsterite
fayalite
clinoferrosilite
clinoenstatite
ferrosilite
enstatite
wollastonite
titanite
diopside
hypersthene
olivine

And everything else:

```
fig, ax = compare_NORMs(df.loc[~volcanic_filter, :], NORM.loc[~volcanic_filter])
plt.show()
```


quartz
zircon
potassium metasilicate
anorthite
sodium metasilicate
acmite
albite
orthoclase
nepheline
leucite
dicalcium silicate
kaliophilite
apatite
chromite
ilmenite
corundum
magnetite
forsterite
fayalite
clinoferrosilite
clinoenstatite
ferrosilite
enstatite
wollastonite
titanite
diopside
hypersthene
olivine

References

Total running time of the script: (0 minutes 1.738 seconds)

4.5.9 Lattice Strain Calculations

> **Note**
>
> This example follows that given during a Institute of Advanced Studies Masterclass with Jon Blundy at the University of Western Australia on the 29th April 2019, and is used here with permission.

Pyrolite includes a function for calculating relative lattice strain[1], which together with the tables of Shannon ionic radii and Young's modulus approximations for silicate and oxide cationic sites enable relatively simple calculation of ionic partitioning in common rock forming minerals.

This example below uses previously characterised calcium and sodium partition coefficients between plagioclase ($CaAl_2Si_2O_8 - NaAlSi_3O8$) and silicate melt to estimate partitioning for other cations based on their ionic radii.

A model parameterised using sodium and calcium partition coefficients[2] is then used to estimate the partitioning for lanthanum into the trivalent site (largely occupied by Al^{3+}), and extended to other trivalent cations (here, the Rare Earth Elements). The final section of the example highlights the mechanism which generates plagioclase's hallmark 'europium anomaly', and the effects of variable europium oxidation state on bulk europium partitioning.

```
import matplotlib.pyplot as plt
import numpy as np

from pyrolite.geochem.ind import REE, get_ionic_radii
from pyrolite.mineral.lattice import strain_coefficient
```

First, we need to define some of the necessary parameters including temperature, the Young's moduli for the X^{2+} and X^{3+} sites in plagioclase (E_2, E_3), and some reference partition coefficients and radii for calcium and sodium:

```
D_Na = 1.35  # Partition coefficient Plag-Melt
D_Ca = 4.1  # Partition coefficient Plag-Melt
Tc = 900  # Temperature, °C
Tk = Tc + 273.15  # Temperature, K
E_2 = 120 * 10**9  # Youngs modulus for 2+ site, Pa
E_3 = 135 * 10**9  # Youngs modulus for 3+ site, Pa
r02, r03 = 1.196, 1.294  # fictive ideal cation radii for these sites
rCa = get_ionic_radii("Ca", charge=2, coordination=8)
rLa = get_ionic_radii("La", charge=3, coordination=8)
```

We can calculate and plot the partitioning of X^{2+} cations relative to Ca^{2+} at a given temperature using their radii and the lattice strain function:

```
fontsize = 8
fig, ax = plt.subplots(1)
ax.set(ylabel="$D_X$", xlabel=r"Radii ($\AA$)", yscale="log")
site2labels = ["Na", "Ca", "Eu", "Sr"]
# get the Shannon ionic radii for the elements in the 2+ site
site2radii = [
```

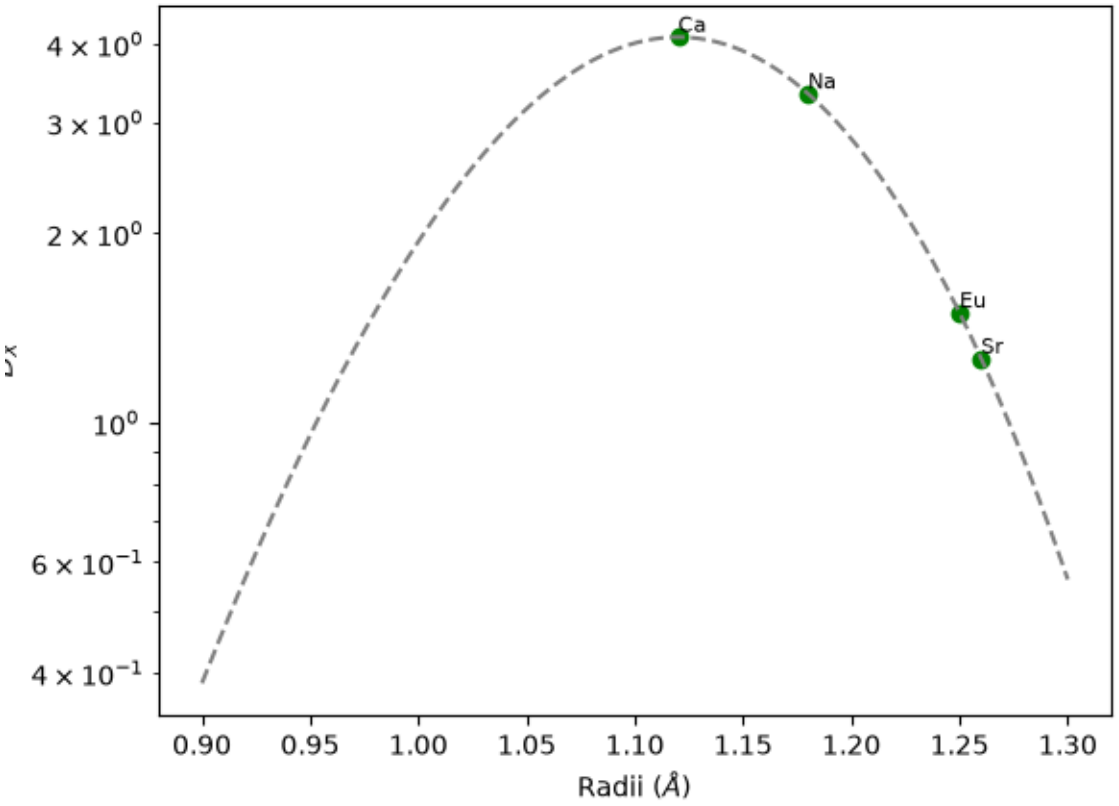
(continues on next page)

[1] Blundy, J., Wood, B., 1994. Prediction of crystal–melt partition coefficients from elastic moduli. Nature 372, 452. doi: 10.1038/372452A0

[2] Dohmen, R., Blundy, J., 2014. A predictive thermodynamic model for element partitioning between plagioclase and melt as a function of pressure, temperature and composition. American Journal of Science 314, 1319–1372. doi: 10.2475/09.2014.04

(continued from previous page)

```
    get_ionic_radii("Na", charge=1, coordination=8),
    *get_ionic_radii(["Ca", "Eu", "Sr"], charge=2, coordination=8),
]
# plot the relative paritioning curve for cations in the 2+ site
site2Ds = D_Ca * np.array(
    [strain_coefficient(rCa, rx, r0=r02, E=E_2, T=Tk) for rx in site2radii]
)
ax.scatter(site2radii, site2Ds, color="g", label="$X^{2+}$ Cations")
# create an index of radii, and plot the relative paritioning curve for the site
xs = np.linspace(0.9, 1.3, 200)
curve2Ds = D_Ca * strain_coefficient(rCa, xs, r0=r02, E=E_2, T=Tk)
ax.plot(xs, curve2Ds, color="0.5", ls="--")
# add the element labels next to the points
for l, r, d in zip(site2labels, site2radii, site2Ds):
    ax.annotate(
        l, xy=(r, d), xycoords="data", ha="left", va="bottom", fontsize=fontsize
    )
fig
```




```
<Figure size 640x480 with 1 Axes>
```

When it comes to estimating the partitioning of X^{3+} cations, we'll need a reference point - here we'll use D_{La} to calculate relative partitioning of the other Rare Earth Elements, although you may have noticed it is not defined above.

Through a handy relationship, we can estimate D_{La} based on the easier measured D_{Ca}, D_{Na} and temperature[Page 85, 2]:

```
D_La = (D_Ca**2 / D_Na) * np.exp((529 / Tk) - 3.705)
D_La  # 0.48085
```

```
np.float64(0.48084614946362086)
```

Now D_{La} is defined, we can use it as a reference for the other REE:

```
site3labels = REE(dropPm=True)
# get the Shannon ionic radii for the elements in the 3+ site
site3radii = get_ionic_radii([x for x in REE(dropPm=True)], charge=3, coordination=8)
site3Ds = D_La * np.array(
    [strain_coefficient(rLa, rx, r0=r03, E=E_3, T=Tk) for rx in site3radii]
)
# plot the relative paritioning curve for cations in the 3+ site
ax.scatter(site3radii, site3Ds, color="purple", label="$X^{3+}$ Cations")
# plot the relative paritioning curve for the site
curve3Ds = D_La * strain_coefficient(rLa, xs, r0=r03, E=E_3, T=Tk)
ax.plot(xs, curve3Ds, color="0.5", ls="--")
# add the element labels next to the points
for l, r, d in zip(site3labels, site3radii, site3Ds):
    ax.annotate(
        l, xy=(r, d), xycoords="data", ha="right", va="bottom", fontsize=fontsize
    )
fig
```

```
<Figure size 640x480 with 1 Axes>
```

As europium is commonly present as a mixture of both Eu^{2+} and Eu^{3+}, the effective partitioning of Eu will be intermediate between that of $D_{Eu^{2+}}`and : math : `D_{Eu^{3+}}$. Using a 60:40 mixture of $Eu^{3+} : Eu^{2+}$ as an example, this effective partition coefficient can be calculated:

```
X_Eu3 = 0.6
# calculate D_Eu3 relative to D_La
D_Eu3 = D_La * strain_coefficient(
    rLa, get_ionic_radii("Eu", charge=3, coordination=8), r0=r03, E=E_3, T=Tk
)
# calculate D_Eu2 relative to D_Ca
D_Eu2 = D_Ca * strain_coefficient(
    rCa, get_ionic_radii("Eu", charge=2, coordination=8), r0=r02, E=E_2, T=Tk
)
# calculate the effective parition coefficient
D_Eu = (1 - X_Eu3) * D_Eu2 + X_Eu3 * D_Eu3
# show the effective partition coefficient relative to the 2+ and 3+ endmembers
radii, ds = (
    [get_ionic_radii("Eu", charge=c, coordination=8) for c in [3, 3, 2, 2]],
    [D_Eu3, D_Eu, D_Eu, D_Eu2],
)
ax.plot(radii, ds, ls="--", color="0.9", label="Effective $D_{Eu}$", zorder=-1)
ax.legend(bbox_to_anchor=(1.05, 1))
fig
```

```
<Figure size 640x480 with 1 Axes>
```

Fitting Lattice Strain Models

Given the lattice strain model and a partitioning profile (for e.g. REE data), we can also fit a model to a given curve; here we fit to our REE data above, for which we have some known parameters to compare to. Note that this uses the youngs modulus approximation behind the scenes where it isn't provided:

```
from pyrolite.mineral.lattice import fit_lattice_strain, youngs_modulus_approximation

t0 = 273.15 + 700  # estimated temperature
_ri, _tk, _D = fit_lattice_strain(site3radii, site3Ds, z=3, t0=t0)
```

We can compare the results of this fit to our source parameters - the ionic radius of La, 900°C and estimated D_{La} - note that the temperature isn't being fit well here, but the other parameters are recovered reasonably well:

```
import pandas as pd

pd.DataFrame(
    [(_ri, rLa), (Tk, _tk), (D_La, _D)],
    index=["radii", "T", "D"],
    columns=["Source Parameters", "Fit Parameters"],
).T
```

From this point, we could We can also compare the curves visually:

```
from pyrolite.plot.spider import REE_v_radii

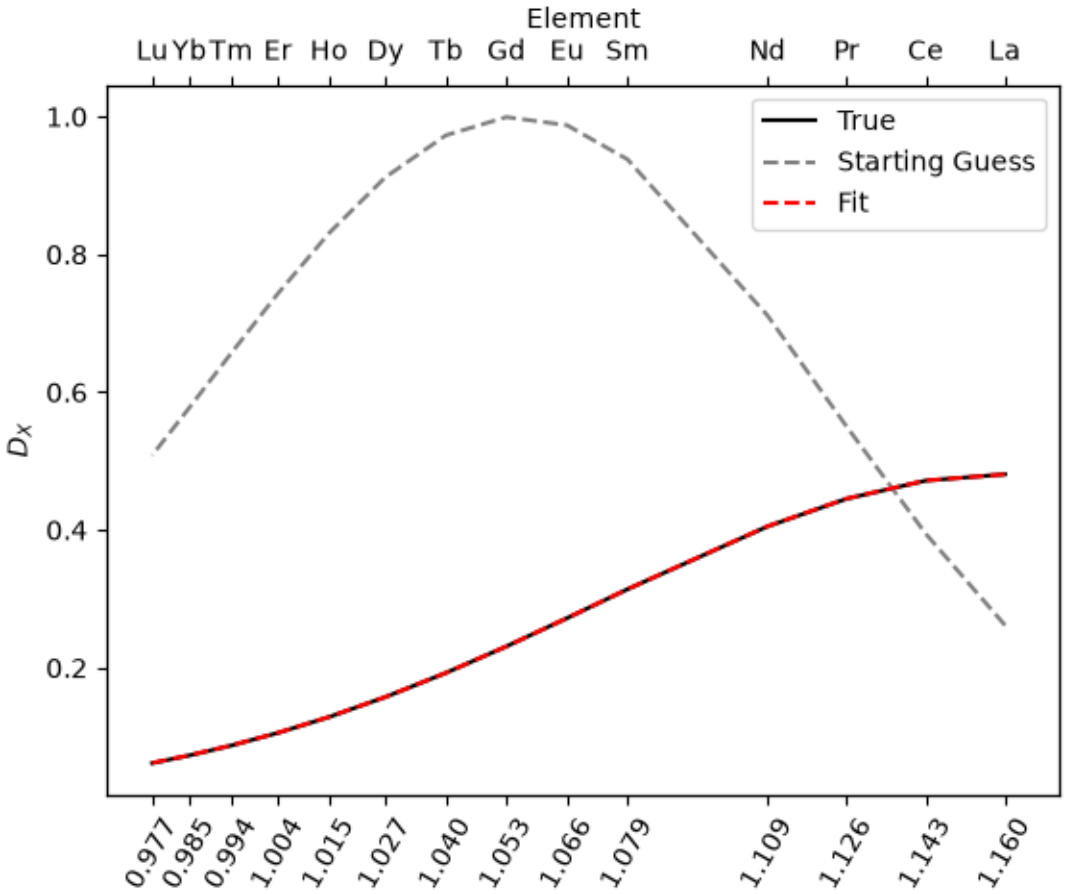
ax = REE_v_radii(index="radii")
ax.set(ylabel="$D_X$", xlabel=r"Radii ($\AA$)")

ax.plot(site3radii, site3Ds, label="True", color="k")

r0 = site3radii.mean()
starting_guess = strain_coefficient(
    r0, site3radii, r0=r0, z=3, T=t0, E=youngs_modulus_approximation(3, r0)
)
ax.plot(site3radii, starting_guess, label="Starting Guess", ls="--", color="0.5")

fit_curve = _D * strain_coefficient(
    _ri, site3radii, r0=_ri, T=_tk, z=3, E=youngs_modulus_approximation(3, _ri)
)
ax.plot(site3radii, fit_curve, color="r", ls="--", label="Fit")

ax.legend()
ax.figure
```




```
<Figure size 640x480 with 2 Axes>
```

See also

Examples:
Shannon Radii, REE Radii Plot

Functions:
strain_coefficient(), *youngs_modulus_approximation()*, *fit_lattice_strain()*, get_ionic_radii()

References

Total running time of the script: (0 minutes 0.200 seconds)

4.5.10 lambdas: Parameterising REE Profiles

Orthogonal polynomial decomposition can be used for dimensional reduction of smooth function over an independent variable, producing an array of independent values representing the relative weights for each order of component polynomial. This is an effective method to parameterise and compare the nature of smooth profiles.

In geochemistry, the most applicable use case is for reduction Rare Earth Element (REE) profiles. The REE are a collection of elements with broadly similar physicochemical properties (the lanthanides), which vary with ionic radii. Given their similar behaviour and typically smooth function of normalised abundance vs. ionic radii, the REE profiles

and their shapes can be effectively parameterised and dimensionally reduced (14 elements summarised by 3-4 shape parameters).

> **Note**
>
> A publication discussing this implementation of *lambdas* together with fitting tetrads for REE patterns has been published in Mathematical Geosciences!

Here we generate some example data, reduce these to lambda values, and visualise the results.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


np.random.seed(82)
```

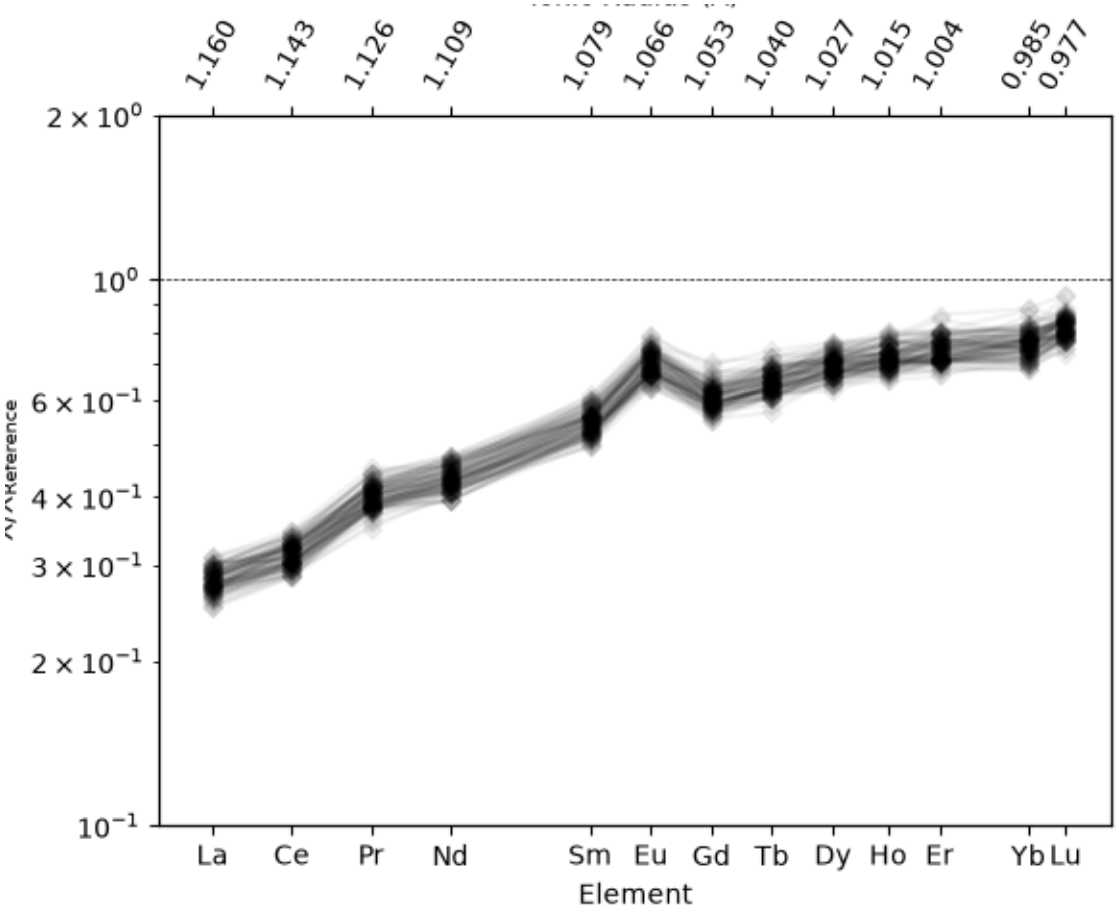
First we'll generate some example synthetic data based around Depleted MORB Mantle:

```
from pyrolite.util.synthetic import example_spider_data

df = example_spider_data(
    noise_level=0.05,
    size=100,
    start="DMM_WH2005",
    norm_to="Chondrite_PON",
    offsets={"Eu": 0.2},
).pyrochem.REE.pyrochem.denormalize_from("Chondrite_PON")
df.head(2)
```

Let's have a quick look at what this REE data looks like normalized to Primitive Mantle:

```
df.pyrochem.normalize_to("PM_PON").pyroplot.REE(alpha=0.05, c="k", unity_line=True)
plt.show()
```



From this REE data we can fit a series of orthogonal polynomials, and subsequently used the regression coefficients ('lambdas') as a parameterisation of the REE pattern/profile:

```
ls = df.pyrochem.lambda_lnREE(degree=4)
ls.head(2)
```

So what's actually happening here? To get some idea of what these coefficients correspond to, we can pull this process apart and visualise our REE profiles as the sum of the series of orthogonal polynomial components of increasing order. As lambdas represent the coefficients for the regression of log-transformed normalised data, to compare the polynomial components and our REE profile we'll first need to normalize it to the appropriate composition (here `"ChondriteREE_ON"`) before taking the logarithm.

With our data, we've then fit a function of ionic radius with the form $f(r) = \lambda_0 + \lambda_1 f_1 + \lambda_2 f_2 + \lambda_3 f_3 ...$ where the polynomial components of increasing order are $f_1 = (r - \beta_0)$, $f_2 = (r - \gamma_0)(r - \gamma_1)$, $f_3 = (r - \delta_0)(r - \delta_1)(r - \delta_2)$ and so on. The parameters β, γ, δ are pre-computed such that the polynomial components are indeed independent. Here we can visualise how these polynomial components are summed to produce the regressed profile, using the last REE profile we generated above as an example:

```
from pyrolite.util.lambdas.plot import plot_lambdas_components

ax = (
    df.pyrochem.normalize_to("ChondriteREE_ON")
    .iloc[-1, :]
    .apply(np.log)
    .pyroplot.REE(color="k", label="Data", logy=False)
```

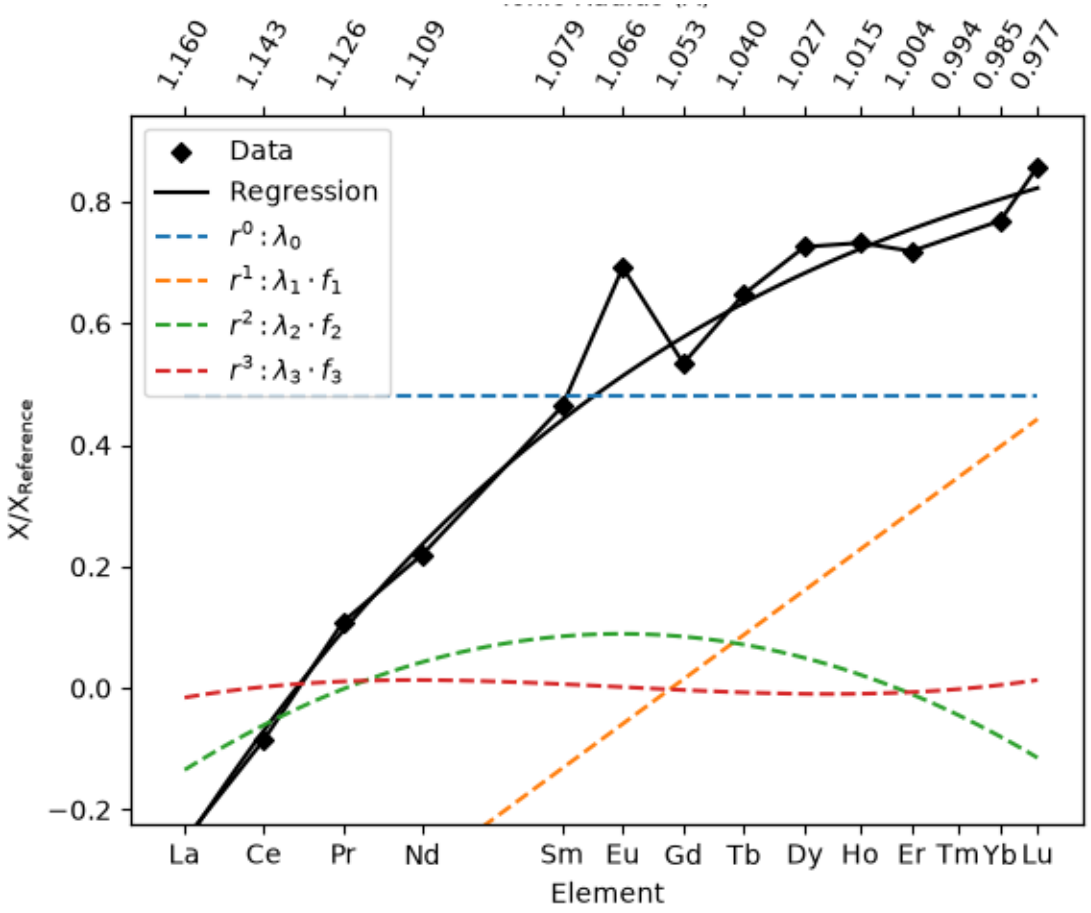
(continues on next page)

(continued from previous page)

```
)

plot_lambdas_components(ls.iloc[-1, :], ax=ax)

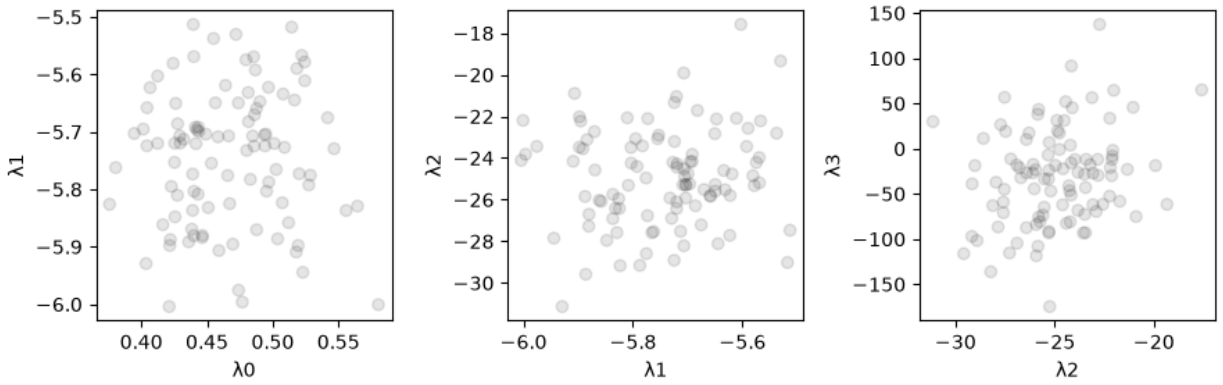
ax.legend()
plt.show()
```



Now that we've gone through a brief introduction to how the lambdas are generated, let's quickly check what the coefficient values themselves look like:

```
fig, ax = plt.subplots(1, 3, figsize=(9, 3))
for ix in range(ls.columns.size - 1):
    ls[ls.columns[ix : ix + 2]].pyroplot.scatter(ax=ax[ix], alpha=0.1, c="k")

plt.tight_layout()
```



But what do these parameters correspond to? From the deconstructed orthogonal polynomial above, we can see that λ_0 parameterises relative enrichment (this is the mean value of the logarithm of Chondrite-normalised REE abundances), λ_1 parameterises a linear slope (here, LREE enrichment), and higher order terms describe curvature of the REE pattern. Through this parameterisation, the REE profile can be effectively described and directly linked to geochemical processes. While the amount of data we need to describe the patterns is lessened, the values themselves are more meaningful and readily used to describe the profiles and their physical significance.

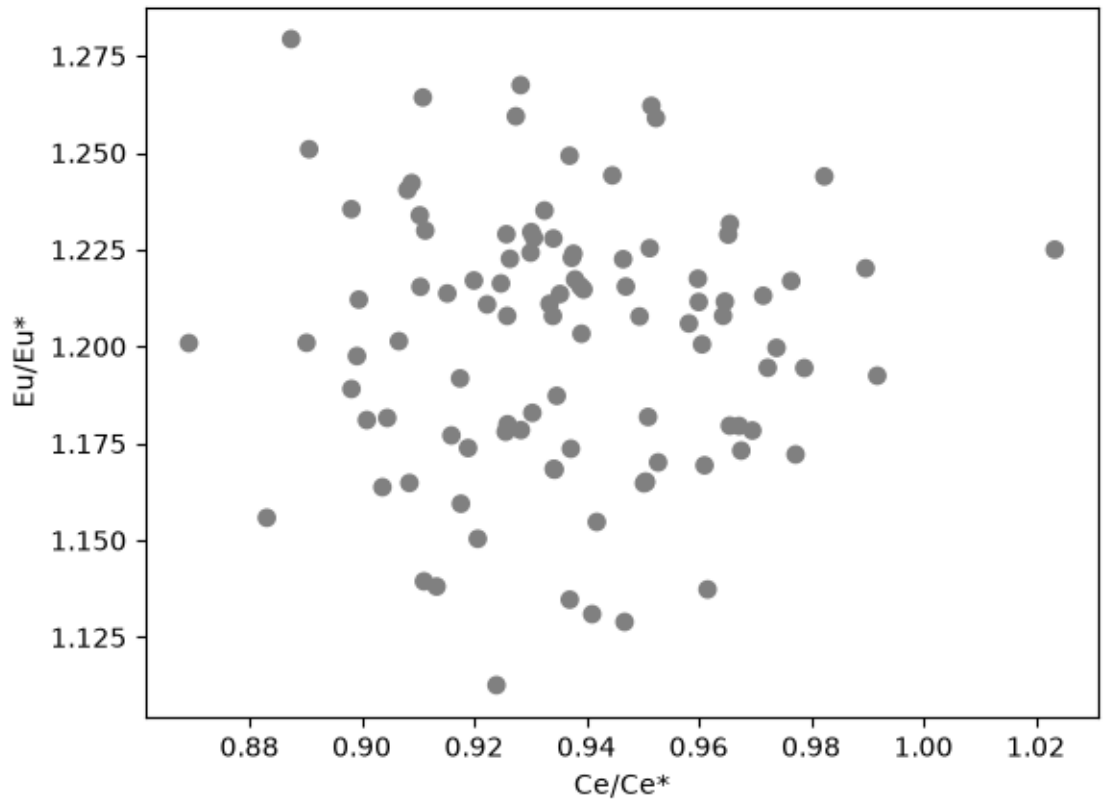
The visualisation of λ_1-λ_2 can be particularly useful where you're trying to compare REE profiles.

We've used a synthetic dataset here which is by design approximately normally distributed, so the values themeselves here are not particularly revealing, but they do illustrate the expected mangitudes of values for each of the parameters.

Dealing With Anomalies

Note that we've not used Eu in this regression - Eu anomalies are a deviation from the 'smooth profile' we need to use this method. Consider this if your data might also exhibit significant Ce anomalies, you might need to exclude this data. For convenience there is also functionality to calculate anomalies derived from the orthogonal polynomial fit itself (rather than linear interpolation methods). Below we use the `anomalies` keyword argument to also calculate the $\frac{Ce}{Ce*}$ and $\frac{Eu}{Eu*}$ anomalies (note that these are excluded from the fit):

```
ls_anomalies = df.pyrochem.lambda_lnREE(anomalies=["Ce", "Eu"])
ax = ls_anomalies.iloc[:, -2:].pyroplot.scatter(color="0.5")
plt.show()
```



Coefficient Uncertainties and Fit Quality

In order to determine the relative significance of the parameterisation and 'goodness of fit', the functions are able to estimate uncertainties on the returned coefficients (lambdas and taus) and will also return the chi-square value (χ^2; equivalent to the MSWD) where requested. This will be appended to the end of the dataframe. Note that if you do not supply an estimate of observed value uncertainties a default of 1% of the log-mean will be used.

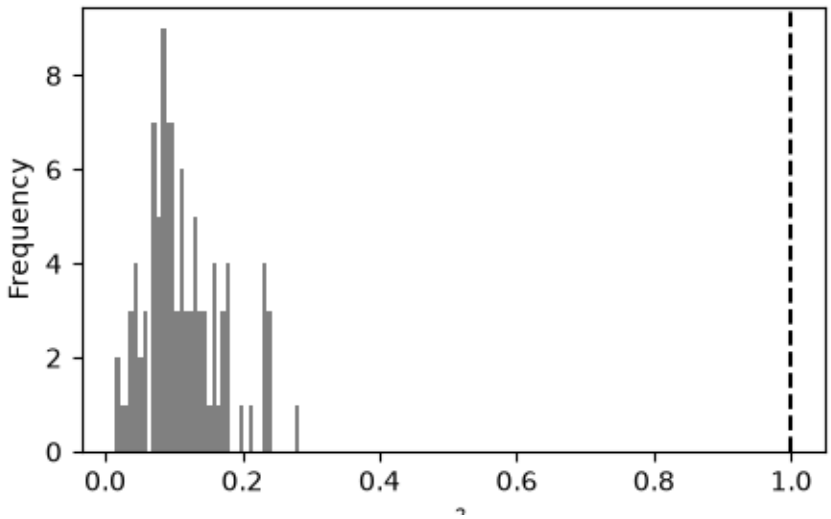
To append the reduced chi-square for each row, the keyword argument `add_X2=True` can be used; here we've estimated 10% uncertainty on the REE:

```
ls = df.pyrochem.lambda_lnREE(add_X2=True, sigmas=0.1, anomalies=["Eu", "Ce"])
ls.columns
```

```
Index(['0', '1', '2', '3', 'X2', 'Eu/Eu*', 'Ce/Ce*'], dtype='str')
```

We can have a quick look at the χ^2 values look like for the synthetic dataset, given the assumed 10% uncertainties. While the fit appears reasonable for a good fraction of the dataset (~2 and below), for some rows it is notably worse:

```
fig, ax = plt.subplots(1, figsize=(5, 3))
ax = ls["X2"].plot.hist(ax=ax, bins=40, color="0.5")
ax.set(xlabel=r"$\chi^2$")
ax.axvline(1, color="k", ls="--")
plt.show()
```



We can also examine the estimated uncertainties on the coefficients from the fit by adding the keyword argument `add_uncertainties=True` (note: these do not explicitly propagate observation uncertainties):

```
ls = df.pyrochem.lambda_lnREE(add_uncertainties=True)
ls.columns
```

```
Index(['0', '1', '2', '3', '0_', '1_', '2_', '3_'], dtype='str')
```

Notably, on the scale of natural dataset variation, these uncertainties may end up being smaller than symbol sizes. If your dataset happened to have a great deal more noise, you may happen to see them - for demonstration purposes we can generate a noiser dataset and have a quick look at what these uncertainties *could* look like:

```
ls = (df + 3 * np.exp(np.random.randn(*df.shape))).pyrochem.lambda_lnREE(
    add_uncertainties=True
)
```

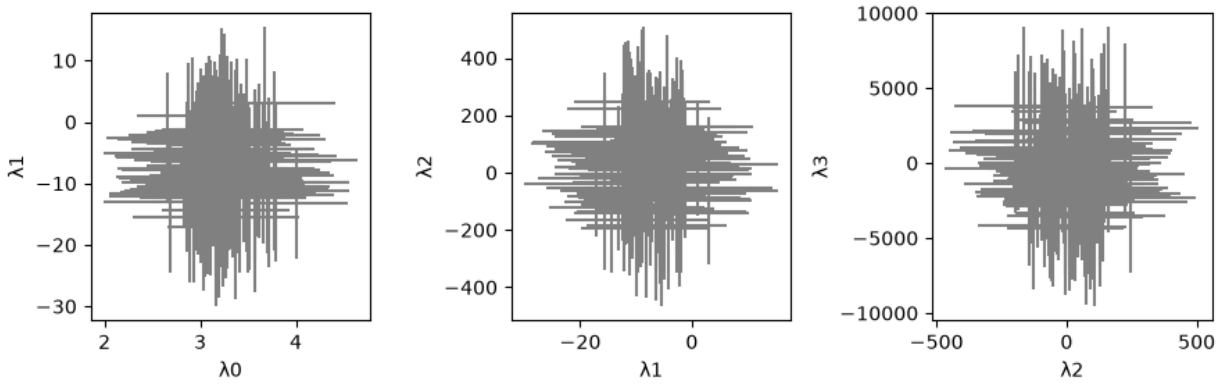
With this 'noisy' dataset, we can see some of the errorbars:

```
fig, ax = plt.subplots(1, 3, figsize=(9, 3))
ax = ax.flat
dc = ls.columns.size // 2
for ix, a in enumerate(ls.columns[:3]):
    i0, i1 = ix, ix + 1
    ax[ix].set(xlabel=ls.columns[i0], ylabel=ls.columns[i1])
    ax[ix].errorbar(
        ls.iloc[:, i0],
        ls.iloc[:, i1],
        xerr=ls.iloc[:, i0 + dc] * 2,
        yerr=ls.iloc[:, i1 + dc] * 2,
        ls="none",
        ecolor="0.5",
        markersize=1,
        color="k",
    )
plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



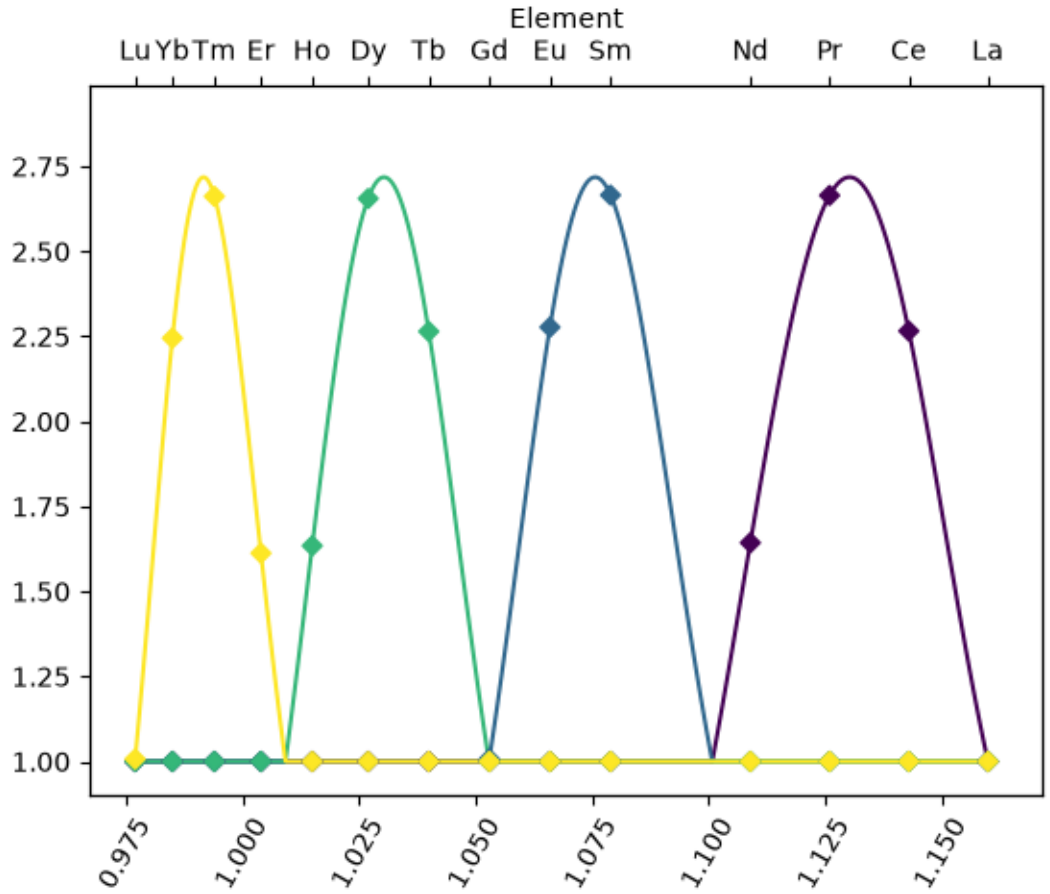
Fitting Tetrads

In addition to fitting orthogonal polynomial functions, the ability to fit tetrad functions has also recently been added. This supplements the λ coefficients with τ coefficients which describe subtle electronic configuration effects affecting sequential subsets of the REE. Below we plot four profiles - each describing only a single tetrad - to illustrate the shape of these function components. Note that these are functions of z, and are here transformed to plot against radii.

```
from pyrolite.util.lambdas.plot import plot_profiles

# let's first create some synthetic pattern parameters
# we want lambdas to be zero, and each of the tetrads to be shown in only one pattern
lambdas = np.zeros((4, 5))
tetrads = np.eye(4)
# putting it together to generate four sets of combined parameters
fit_parameters = np.hstack([lambdas, tetrads])

ax = plot_profiles(
    fit_parameters,
    tetrads=True,
    color=np.arange(4),
)
plt.show()
```



In order to also fit these function components, you can pass the keyword argument `fit_tetrads=True` to *`lambda_lnREE()`* and related functions:

```
lts = df.pyrochem.lambda_lnREE(degree=4, fit_tetrads=True)
```

We can see that the four extra τ Parameters have been appended to the right of the lambdas within the output:

```
lts.head(2)
from pyrolite.geochem.ind import REE
```

Below we'll look at some of the potential issues of fitting lambdas and tetrads together - by examining the effects of i) fitting tetrads where there are none and ii) not fitting tetrads where they do indeed exist using some synthetic datasets.

```
from pyrolite.util.synthetic import example_patterns_from_parameters

ls = np.array(
    [
        [2, 5, -30, 100, -600, 0, 0, 0, 0],  # lambda-only
        [3, 15, 30, 300, 1500, 0, 0, 0, 0],  # lambda-only
        [1, 5, -50, 0, -1000, -0.3, -0.7, -1.4, -0.2],  # W-pattern tetrad
        [5, 15, 50, 400, 2000, 0.6, 1.1, 1.5, 0.3],  # M-pattern tetrad
    ]
)
# now we use these parameters to generate some synthetic log-scaled normalised REE
```

(continues on next page)

(continued from previous page)

```
# patterns and add a bit of noise
pattern_df = pd.DataFrame(
    np.vstack([example_patterns_from_parameters(l, includes_tetrads=True) for l in ls]),
    columns=REE(),
)
# We can now fit these patterns and see what the effect of fitting and not-Fitting
# tetrads might look like in these (slightly extreme) cases:
fit_ls_only = pattern_df.pyrochem.lambda_lnREE(
    norm_to=None, degree=5, fit_tetrads=False
)
fit_ts = pattern_df.pyrochem.lambda_lnREE(norm_to=None, degree=5, fit_tetrads=True)
```

We can now examine what the differences between the fits are. Below we plot the four sets of synthetic REE patterns (lambda-only above and lamba+tetrad below) and examine the relative accuracy of fitting some of the higher order lambda parameters where tetrads are also fit:

```
from pyrolite.util.plot.axes import share_axes

x, y = 2, 3
categories = np.repeat(np.arange(ls.shape[0]), 100)
colors = np.array([str(ix) * 2 for ix in categories])
l_only = categories < 2

ax = plt.figure(figsize=(12, 7)).subplot_mosaic(
    """
    AAABBCC
    DDDEEFF
    """
)
share_axes([ax["A"], ax["D"]])
share_axes([ax["B"], ax["C"], ax["E"], ax["F"]])

ax["B"].set_title("lambdas only Fit")
ax["C"].set_title("lambdas+tetrads Fit")


for a, fltr in zip(["A", "D"], [l_only, ~l_only]):
    pattern_df.iloc[fltr, :].pyroplot.spider(
        ax=ax[a],
        label="True",
        unity_line=True,
        alpha=0.5,
        color=colors[fltr],
    )

for a, fltr in zip(["B", "E"], [l_only, ~l_only]):
    fit_ls_only.iloc[fltr, [x, y]].pyroplot.scatter(
        ax=ax[a],
        alpha=0.2,
        color=colors[fltr],
    )
```

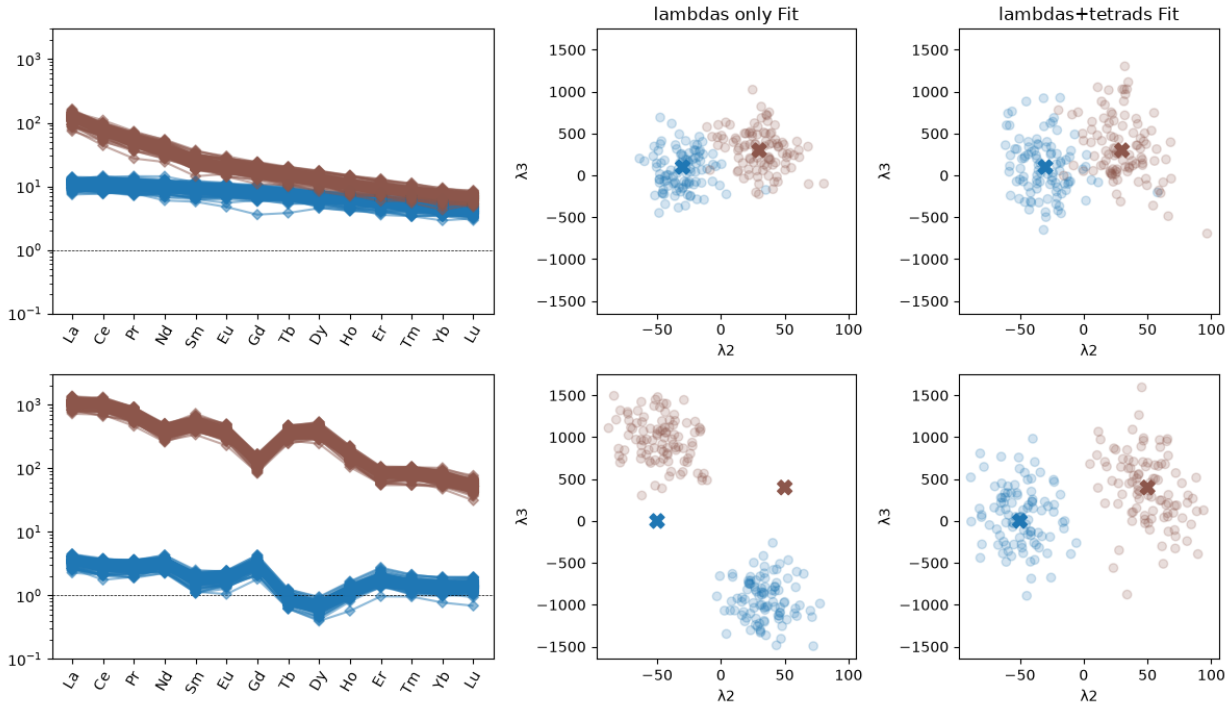
(continues on next page)

(continued from previous page)

```
for a, fltr in zip(["C", "F"], [l_only, ~l_only]):
    fit_ts.iloc[fltr, [x, y]].pyroplot.scatter(
        ax=ax[a],
        alpha=0.2,
        color=colors[fltr],
    )


true = pd.DataFrame(ls[:, [x, y]], columns=[fit_ls_only.columns[ix] for ix in [x, y]])
for ix, a in enumerate(["B", "C", "E", "F"]):
    true.iloc[np.array([ix < 2, ix < 2, ix >= 2, ix >= 2]), :].pyroplot.scatter(
        ax=ax[a],
        color=np.array([str(ix) * 2 for ix in np.arange(ls.shape[0] // 2)]),
        marker="X",
        s=100,
    )

plt.tight_layout()
plt.show()
```



More Advanced Customisation

Above we've used default parameterisations for calculating *lambdas*, but `pyrolite` allows you to customise the parameterisation of both the orthogonal polynomial components used in the fitting process as well as what data and algorithm is used in the fit itself.

To exclude some elements from the *fit* (e.g. Eu which is excluded by default, and and potentially Ce), you can either i) filter the dataframe such that the columns aren't passed, or ii) explicitly exclude them:

```
# filtering the dataframe first
target_columns = [i for i in df.columns if i not in ["Eu", "Ce"]]
ls_noEuCe_filtered = df[target_columns].pyrochem.lambda_lnREE()
# excluding some elements
ls_noEuCe_excl = df.pyrochem.lambda_lnREE(exclude=["Eu", "Ce"])

# quickly checking equivalence
np.allclose(ls_noEuCe_excl, ls_noEuCe_filtered)
```

```
True
```

While the results should be numerically equivalent, `pyrolite` does provide two algorithms for fitting lambdas. The first follows almost exactly the original formulation (`algorithm="ONeill"`; this was translated from VBA), while the second simply uses a numerical optimization routine from `scipy` to achieve the same thing (`algorithm="opt"`; this is a fallback for where singular matricies pop up):

```
# use the original version
ls_linear = df.pyrochem.lambda_lnREE(algorithm="ONeill")

# use the optimization algorithm
ls_opt = df.pyrochem.lambda_lnREE(algorithm="opt")
```

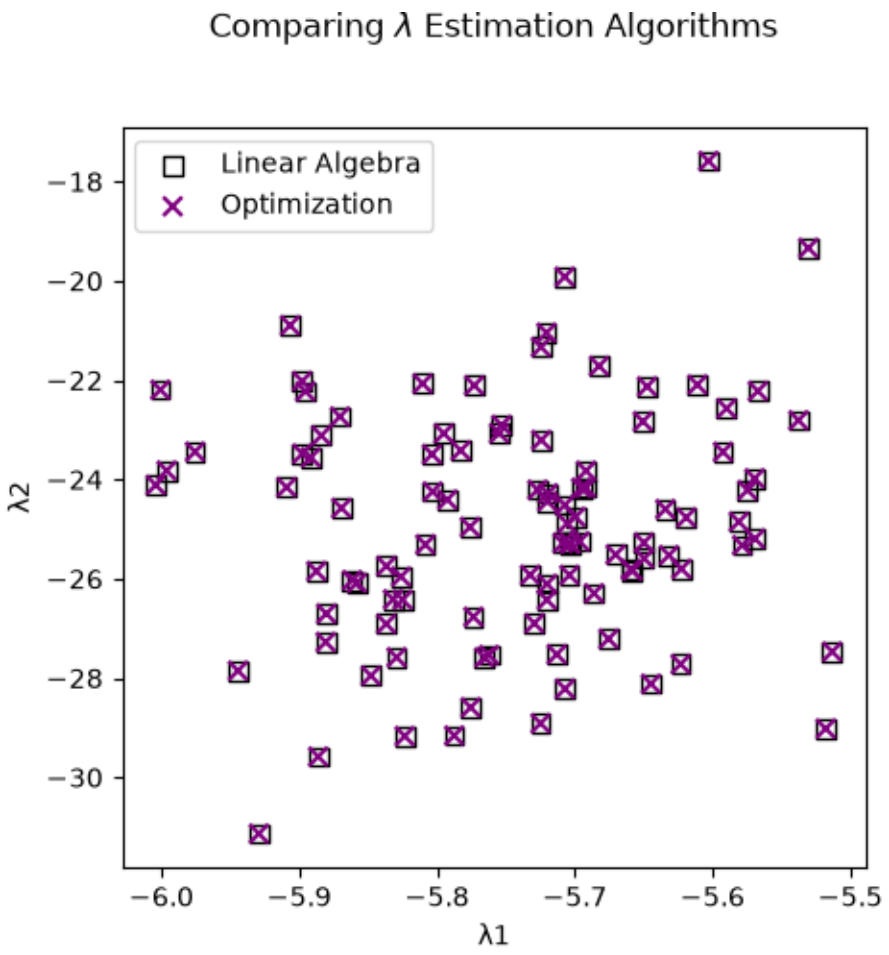
To quickly demonstrate the equivalance, we can check numerically (to within 0.001%):

```
np.allclose(ls_linear, ls_opt, rtol=10e-5)
```

```
True
```

Or simply plot the results from both:

```
fig, ax = plt.subplots(1, figsize=(5, 5))
ax.set_title(r"Comparing $\lambda$ Estimation Algorithms", y=1.1)
ls_linear.iloc[:, 1:3].pyroplot.scatter(
    ax=ax, marker="s", c="k", facecolors="none", s=50, label="Linear Algebra"
)
ls_opt.iloc[:, 1:3].pyroplot.scatter(
    ax=ax, c="purple", marker="x", s=50, label="Optimization"
)
ax.legend()
plt.show()
```



You can also use orthogonal polynomials defined over a different set of REE, by specifying the parameters using the keyword argument *params*:

```
# this is the original formulation from the paper, where Eu is excluded
ls_original = df.pyrochem.lambda_lnREE(params="ONeill2016")

# this uses a full set of REE
ls_fullREE_polynomials = df.pyrochem.lambda_lnREE(params="full")
```

Note that as of `pyrolite` v0.2.8, the original formulation is used by default, but this will cease to be the case as of the following version, where the full set of REE will instead be used to generate the orthogonal polynomials.

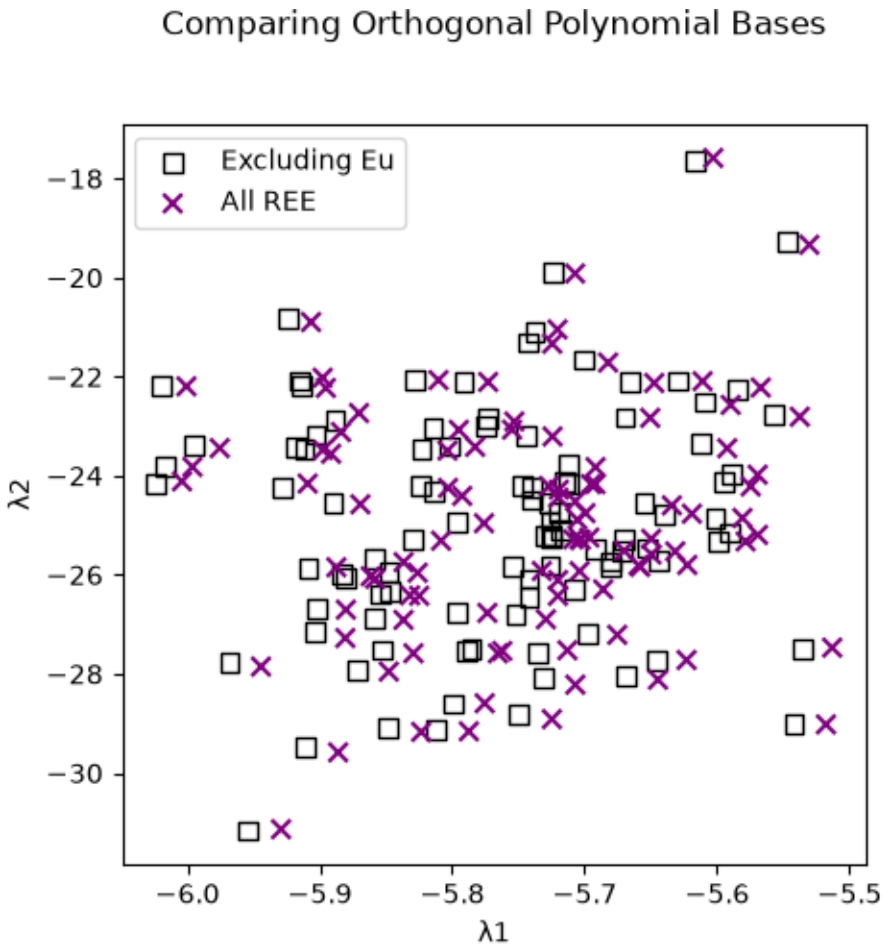
While the results are simlar, there are small differences. They're typically less than 1%:

```
np.abs((ls_original / ls_fullREE_polynomials) - 1).max() * 100
```

```
0    8.818e-01
1    4.396e-01
2    6.742e-01
3    5.221e-07
dtype: float64
```

This can also be visualised:

```
fig, ax = plt.subplots(1, figsize=(5, 5))
ax.set_title("Comparing Orthogonal Polynomial Bases", y=1.1)
ls_original.iloc[:, 1:3].pyroplot.scatter(
    ax=ax, marker="s", c="k", facecolors="none", s=50, label="Excluding Eu"
)
ls_fullREE_polynomials.iloc[:, 1:3].pyroplot.scatter(
    ax=ax, c="purple", marker="x", s=50, label="All REE"
)
ax.legend()
plt.show()
```



References & Citation

For more on using orthogonal polynomials to describe geochemical pattern data, dig into the paper which introduced the method to geochemists:

> O'Neill, H.S.C., 2016. The Smoothness and Shapes of Chondrite-normalized Rare Earth Element Patterns in Basalts. J Petrology 57, 1463–1508. doi: 10.1093/petrology/egw047.

If you're using `pyrolite`'s implementation of *lambdas* in your research, please consider citing a recent publication directly related to this:

> Anenburg, M., & Williams, M. J. (2021). Quantifying the Tetrad Effect, Shape Components, and Ce–Eu–Gd Anomalies in Rare Earth Element Patterns. Mathematical Geosciences. doi: 10.1007/s11004-

021-09959-5

See also

Examples:
Ionic Radii, REE Radii Plot

Functions:
lambda_lnREE(), *get_ionic_radii()*, *pyrolite.plot.pyroplot.REE()*

Total running time of the script: (0 minutes 6.441 seconds)

4.6 Compositional Data Examples

4.6.1 Log-transforms

pyrolite includes a few functions for dealing with compositional data, at the heart of which are i) closure (i.e. everything sums to 100%) and ii) log-transforms to deal with the compositional space. The commonly used log-transformations include the Additive Log-Ratio (*ALR()*), Centred Log-Ratio (*CLR()*), and Isometric Log-Ratio (*ILR()*)[1][2].

This example will show you how to access and use some of these functions in pyrolite.

First let's create some example data:

```
from pyrolite.util.synthetic import normal_frame, random_cov_matrix

df = normal_frame(
    size=100,
    cov=random_cov_matrix(sigmas=[0.1, 0.05, 0.3, 0.6], dim=4, seed=32),
    seed=32,
)
df.describe()
```

Let's have a look at some of the log-transforms, which can be accessed directly from your dataframes (via *pyrolite.comp.pyrocomp*), after you've imported *pyrolite.comp*. Note that the transformations will return *new* dataframes, rather than modify their inputs. For example:

```
lr_df = df.pyrocomp.CLR()  # using a centred log-ratio transformation
```

The transformations are implemented such that the column names generally make it evident which transformations have been applied (here using default simple labelling; see below for other examples):

```
lr_df.columns
```

```
Index(['CLR(SiO2/G)', 'CLR(CaO/G)', 'CLR(MgO/G)', 'CLR(FeO/G)', 'CLR(TiO2/G)'],
      dtype='str')
```

To invert these transformations, you can call the respective inverse transform:

[1] Aitchison, J., 1984. The statistical analysis of geochemical compositions. Journal of the International Association for Mathematical Geology 16, 531–564. doi: 10.1007/BF01029316

[2] Egozcue, J.J., Pawlowsky-Glahn, V., Mateu-Figueras, G., Barceló-Vidal, C., 2003. Isometric Logratio Transformations for Compositional Data Analysis. Mathematical Geology 35, 279–300. doi: 10.1023/A:1023818214614

```
back_transformed = lr_df.pyrocomp.inverse_CLR()
```

Given we haven't done anything to our dataframe in the meantime, we should be back where we started, and our values should all be equal within numerical precision. To verify this, we can use `numpy.allclose()`:

```
import numpy as np

np.allclose(back_transformed, df)
```

```
True
```

In addition to easy access to the transforms, there's also a convenience function for taking a log-transformed mean (log-transforming, taking a mean, and inverse log transforming; `logratiomean()`):

```
df.pyrocomp.logratiomean()
```

```
SiO2    0.241
CaO     0.395
MgO     0.094
FeO     0.104
TiO2    0.166
dtype: float64
```

While this function defaults to using `clr()`, you can specify other log-transforms to use:

```
df.pyrocomp.logratiomean(transform="CLR")
```

```
SiO2    0.241
CaO     0.395
MgO     0.094
FeO     0.104
TiO2    0.166
dtype: float64
```

Notably, however, the logratio means should all give you the same result:

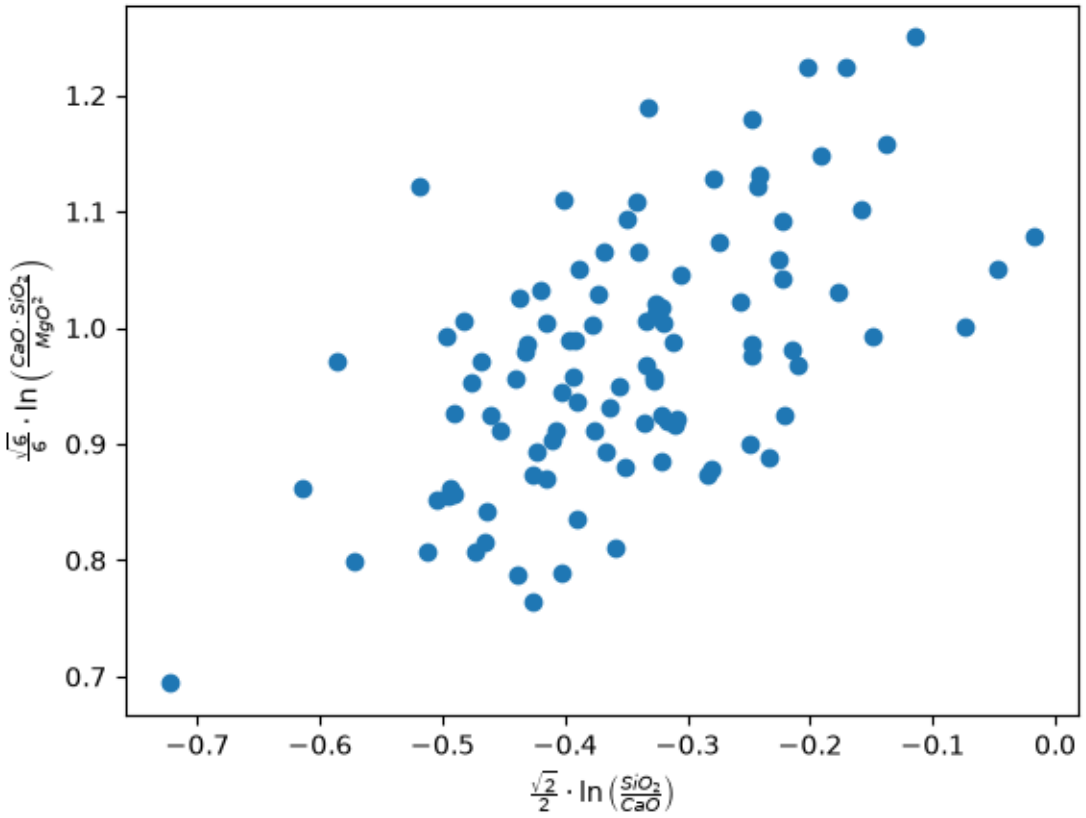
```
np.allclose(
    df.pyrocomp.logratiomean(transform="CLR"),
    df.pyrocomp.logratiomean(transform="ALR"),
) & np.allclose(
    df.pyrocomp.logratiomean(transform="CLR"),
    df.pyrocomp.logratiomean(transform="ILR"),
)
```

```
True
```

To change the default labelling outputs for column names, you can use the *label_mode* parameter, for example to get nice labels for plotting:

```
import matplotlib.pyplot as plt

df.pyrocomp.ILR(label_mode="latex").iloc[:, 0:2].pyroplot.scatter()
plt.show()
```



Alternatively if you simply want numeric indexes which you can use in e.g. a ML pipeline, you can use `label_mode="numeric"`:

```
df.pyrocomp.ILR(label_mode="numeric").columns
```

```
Index(['ILR0', 'ILR1', 'ILR2', 'ILR3'], dtype='str')
```

See also

Examples:
Log Ratio Means, Compositional Data, Ternary Plots

Tutorials:
Ternary Density Plots, Making the Logo

Modules and Functions:
pyrolite.comp.codata, *boxcox()*, *renormalise()*

Total running time of the script: (0 minutes 0.288 seconds)

4.6.2 Spherical Coordinate Transformations

While logratio methods are typically more commonly used to deal with with compositional data, they are not infallible - their principal weakness is being able to deal with true zeros like we have the example dataset below. True zeros are common in mineralogical datasets, and are possible in geochemical datasets to some degree (at least at the level of atom counting..).

An alternative to this is to use a spherical coordinate transform to handle our compositional data. This typically involves treating each covariate as a separate dimension/axis and each composition as a unit vector in this space. The transformation invovles the iterative calculation of an angular representeation of these vectors. N-dimensional vectors transformed to $N-1$ dimensional angular equivalents (as the first angle is that between the first axis and the second). At least two variants of this type of transformation exist - a spherical cosine method and a spherical sine method; these are complementary and the sine method is used here (where angles close to $\pi/2$ represent close to zero abundance, and angles closer to zero/aligning with the repsective component axis represent higher abundances). See below for an example of this demonstrated graphically.

First let's create some example mineralogical abundance data, where at least one of the minerals might occasionally have zero abundance:

```
import numpy as np

from pyrolite.util.synthetic import normal_frame

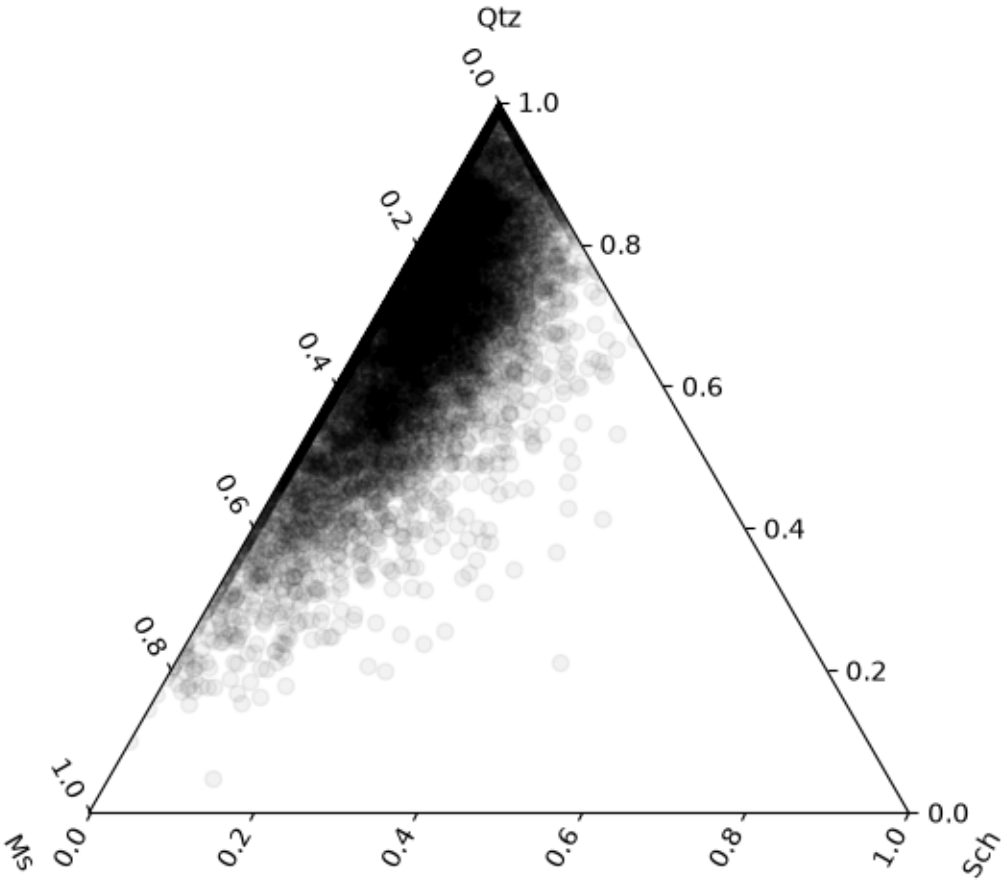
comp = normal_frame(
    columns=["Ab", "Qtz", "Ms", "Sch"],
    mean=[0.5, 1, 0.3, 0.05],
    cov=np.eye(3) * np.array([0.02, 0.5, 0.5]),
    size=10000,
    seed=145,
)
comp += 0.05 * np.random.randn(*comp.shape)
comp[comp <= 0] = 0
comp = comp.pyrocomp.renormalise(scale=1)
```

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:122:␣
↪UserWarning: Non-positive entries found in specified components. Negative values have␣
↪been replaced with NaN. Renormalisation assumes all positive entries.
  warnings.warn(
```

We can quickly visualise this to see that it does indeed have some true zeros:

```
import matplotlib.pyplot as plt

comp[["Qtz", "Ms", "Sch"]].pyroplot.scatter(alpha=0.05, c="k")
plt.show()
```



The spherical coordinate transform functions can be found within *pyrolite.comp.codata*, but can also be accessed from the *pyrocomp* dataframe accessor:

```
angles = comp.pyrocomp.sphere()
angles.head()
```

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:56: UserWarning:␣
↪Non-positive entries found. Closure operation assumes all positive entries.
  warnings.warn(
```

The inverse function can be accessed in the same way:

```
inverted_angles = angles.pyrocomp.inverse_sphere()
inverted_angles.head()
```

We can see that the inverted angles are within precision of the original composition we used:

```
np.isclose(inverted_angles, comp.values).all()
```

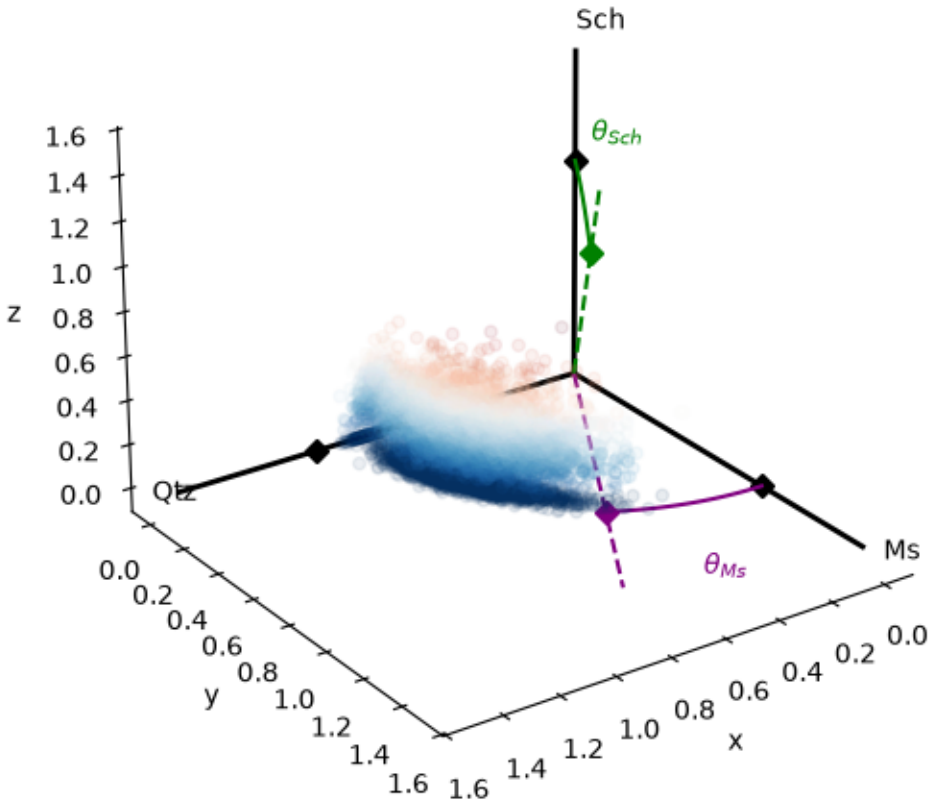
```
np.True_
```

To better understand what's going on here, visualising our data is often the best first step. Below we use a helper fucntion from *pyrolite.util.plot.helpers* to create a 3D axis on which to plot our angular data.

```
from pyrolite.plot.color import process_color
from pyrolite.util.plot.helpers import init_spherical_octant

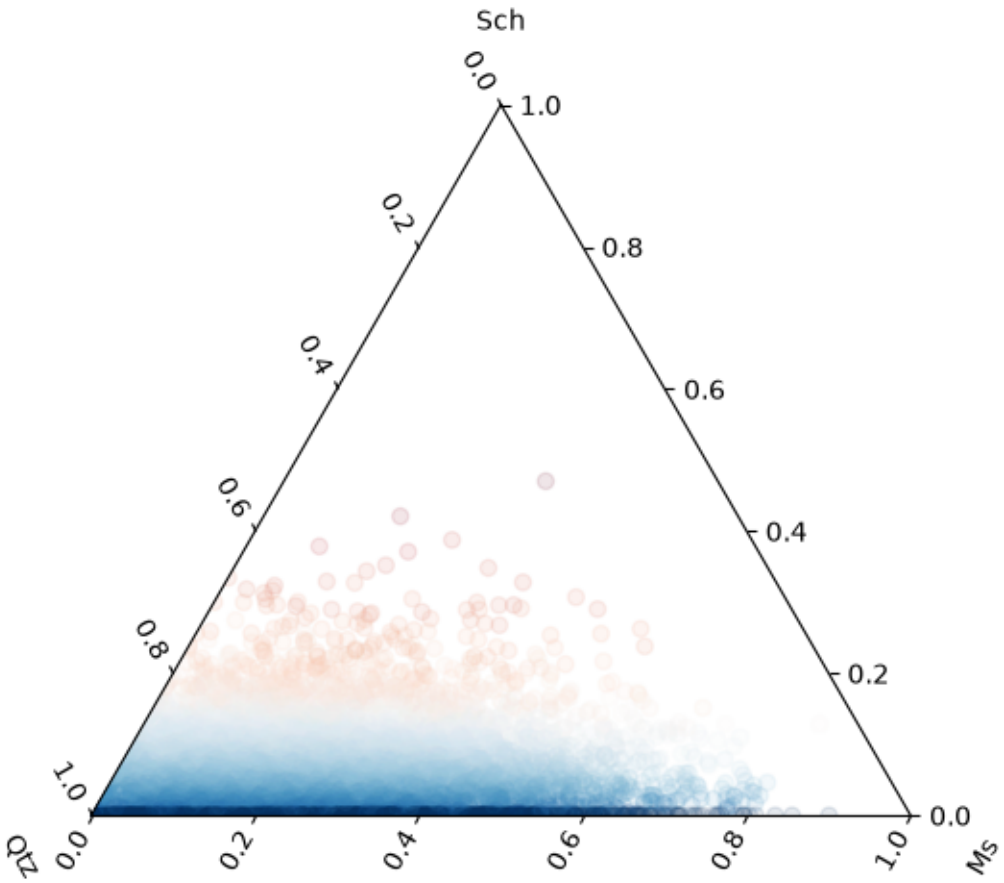
ax = init_spherical_octant(labels=[c[2:] for c in angles.columns], figsize=(6, 6))

# here we can color the points by the angle from the Schorl axis
colors = process_color(angles["_Sch"], cmap="RdBu", alpha=0.1)["c"]
ax.scatter(*np.sqrt(comp.values[:, 1:]).T, c=colors)
plt.show()
```



We can compare this to the equivalent ternary projection of our data; note we need to reorder some columns in order to make this align in the same way:

```
tcolumns = np.array([c[2:] for c in angles.columns])[[2, 0, 1]]
comp[tcolumns].pyroplot.scatter(c=colors)
plt.show()
```



Total running time of the script: (0 minutes 0.819 seconds)

4.6.3 Log Ratio Means

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from pyrolite.comp.codata import ILR, close, inverse_ILR
from pyrolite.util.synthetic import random_cov_matrix


np.random.seed(82)
```

```
def random_compositional_trend(m1, m2, c1, c2, resolution=20, size=1000):
    """
    Generate a compositional trend between two compositions with independent
    variances.
    """
```

(continues on next page)

(continued from previous page)

```
    # generate means intermediate between m1 and m2
    mv = np.vstack([ILR(close(m1)).reshape(1, -1), ILR(close(m2)).reshape(1, -1)])
    ms = np.apply_along_axis(lambda x: np.linspace(*x, resolution), 0, mv)
    # generate covariance matricies intermediate between c1 and c2
    cv = np.vstack([c1.reshape(1, -1), c2.reshape(1, -1)])
    cs = np.apply_along_axis(lambda x: np.linspace(*x, resolution), 0, cv)
    cs = cs.reshape(cs.shape[0], *c1.shape)
    # generate samples from each
    samples = np.vstack(
        [
            np.random.multivariate_normal(m.flatten(), cs[ix], size=size // resolution)
            for ix, m in enumerate(ms)
        ]
    )
    # combine together.
    return inverse_ILR(samples)
```

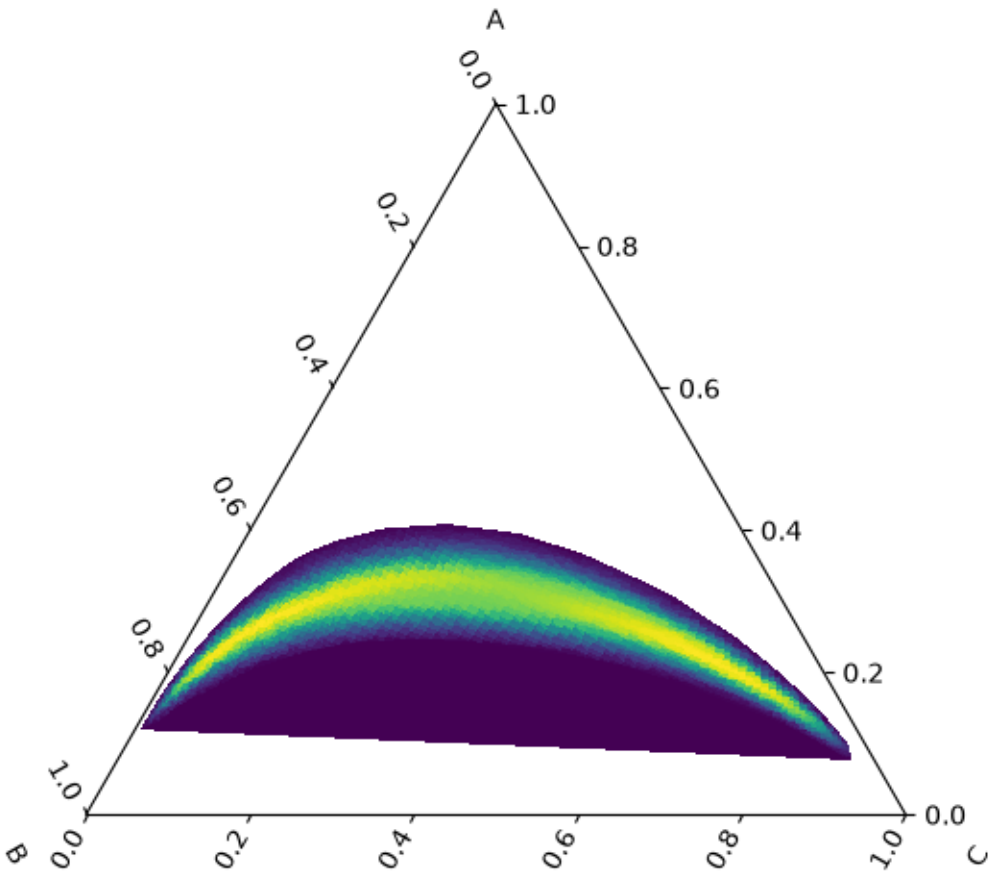
First we create an array of compositions which represent a trend.

```
m1, m2 = np.array([[0.3, 0.1, 2.1]]), np.array([[0.5, 2.5, 0.05]])
c1, c2 = (
    random_cov_matrix(2, sigmas=[0.15, 0.05]),
    random_cov_matrix(2, sigmas=[0.05, 0.2]),
)

trend = pd.DataFrame(
    random_compositional_trend(m1, m2, c1, c2, resolution=100, size=5000),
    columns=["A", "B", "C"],
)
```

We can visualise this compositional trend with a density plot.

```
ax = trend.pyroplot.density(mode="density", bins=100)
plt.show()
```



First we can see where the geometric mean would fall:

```
geomean = trend.mean(axis=0).to_frame().T
ax = geomean.pyroplot.scatter(ax=ax, marker="o", color="r", zorder=2, label="GeoMean")
plt.show()
```

Finally, we can also see where the logratio mean would fall:

```
ILRmean = trend.pyrocomp.logratiomean(transform="ILR")
ax = ILRmean.pyroplot.scatter(ax=ax, color="k", label="LogMean")
plt.show()
```

See also

Examples:
Log Transforms, Compositional Data, Ternary Plots

Tutorials:
Ternary Density Plots, Making the Logo

Modules and Functions:
pyrolite.comp.codata, *renormalise()*

Total running time of the script: (0 minutes 0.709 seconds)

4.6.4 Compositional Data?

pyrolite comes with a few datasets from Aitchison (1984) built in which we can use as examples:

```
import matplotlib.pyplot as plt
import numpy as np

from pyrolite.data.Aitchison import load_kongite

df = load_kongite()
```

For compositional data, everything is relative (thanks to the closure property), so we tend to use ratios to express differences or changes between things. However, if we make incorrect assumptions about the nature of our data, we can get some incorrect answers. Say you want to know the average ratio between A and B:

```
A_on_B = df["A"] / df["B"]
A_on_B.mean()  # 2.8265837788402983
```

```
np.float64(2.8265837788402983)
```

Equally, you could have chosen to calculate the average ratio between B and A

```
B_on_A = df["B"] / df["A"]
B_on_A.mean()  # 0.4709565704852008
```

```
np.float64(0.4709565704852008)
```

You expect these to be invertable, such that A_on_B = 1 / B_on_A; but not so!

```
A_on_B.mean() / (1 / B_on_A.mean())  # 1.3311982026717262
```

```
np.float64(1.3311982026717262)
```

Similarly, the relative variances are different:

```
np.std(A_on_B) / A_on_B.mean()  # 0.6295146309597085
np.std(B_on_A) / B_on_A.mean()  # 0.5020948201979953
```

```
np.float64(0.5020948201979953)
```

This improves when using logratios in place of simple ratios, prior to exponentiating means

```
logA_on_B = (df["A"] / df["B"]).apply(np.log)
logB_on_A = (df["B"] / df["A"]).apply(np.log)
```

The logratios are invertible:

```
np.exp(logA_on_B.mean())  # 2.4213410747400514
1 / np.exp(logB_on_A.mean())  # 2.421341074740052
```

```
np.float64(2.421341074740052)
```

The logratios also have the same variance:

```
(np.std(logA_on_B) / logA_on_B.mean()) ** 2  # 0.36598579018127086
(np.std(logB_on_A) / logB_on_A.mean()) ** 2  # 0.36598579018127086
```

```
np.float64(0.36598579018127086)
```

These peculiarities result from incorrect assumptions regarding the distribution of the data: ratios of compositional components are typically *lognormally* distributed, rather than *normally* distributed, and the compositional components themselves commonly have a Poisson distribution . These distributions contrast significantly with the normal distribution at the core of most statistical tests. We can compare distributions with similar means and variances but different forms, and note that the normal distribution has one immediate failure, in that it has non-zero probability density below 0, and we know that you can't have negative atoms!

```
from scipy.stats import lognorm, norm, poisson

means = [[10, 10], [10, 20], [20, 100], [1000, 50]]
fig, ax = plt.subplots(len(means), 4, figsize=(11, 8))
ax[0, 0].set_title("A")
ax[0, 1].set_title("B")
ax[0, 2].set_title("Normal Fit to B/A")
ax[0, 3].set_title("Lognormal Fit to B/A")
ax[-1, 0].set_xlabel("A")
ax[-1, 1].set_xlabel("B")
ax[-1, 2].set_xlabel("B/A")
ax[-1, 3].set_xlabel("B/A")
for ix, (m1, m2) in enumerate(means):
    p1, p2 = poisson(mu=m1), poisson(mu=m2)
    y1, y2 = p1.rvs(2000), p2.rvs(2000)
    ratios = y2[y1 > 0] / y1[y1 > 0]

    y1min, y1max = y1.min(), y1.max()
    y2min, y2max = y2.min(), y2.max()
    ax[ix, 0].hist(
        y1,
        color="0.5",
        alpha=0.6,
        label="A",
        bins=np.linspace(y1min - 0.5, y1max + 0.5, (y1max - y1min) + 1),
    )
    ax[ix, 1].hist(
        y2,
        color="0.5",
        alpha=0.6,
        label="B",
        bins=np.linspace(y2min - 0.5, y2max + 0.5, (y2max - y2min) + 1),
    )

    # normal distribution fit
    H, binedges, patches = ax[ix, 2].hist(
        ratios, color="Purple", alpha=0.6, label="Ratios", bins=100
    )
    loc, scale = norm.fit(ratios, loc=0)
    pdf = norm.pdf(binedges, loc, scale)
    twin2 = ax[ix, 2].twinx()
```

(continues on next page)

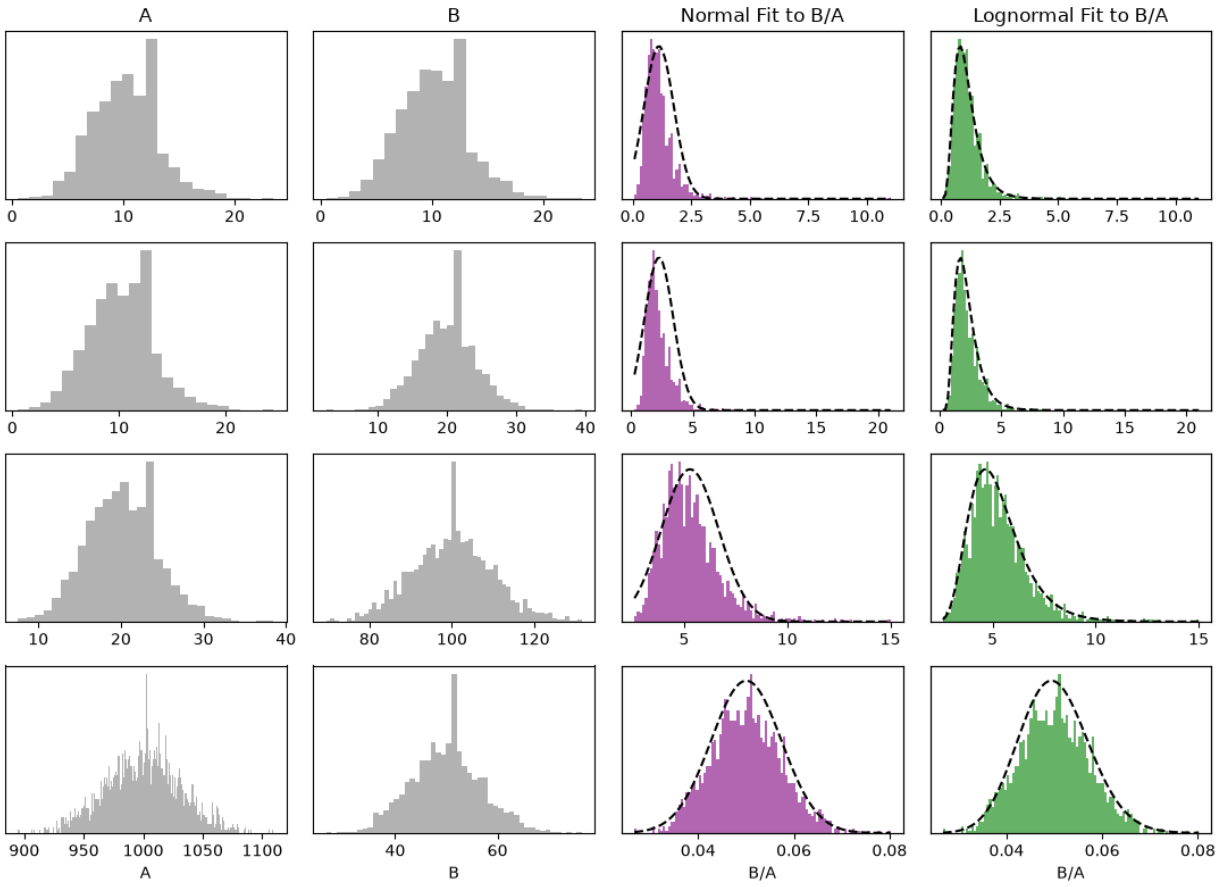
(continued from previous page)

```
    twin2.set_ylim(0, 1.1 * np.max(pdf))
    twin2.plot(binedges, pdf, color="k", ls="--", label="Normal Fit")

    # log-normal distribution fit
    H, binedges, patches = ax[ix, 3].hist(
        ratios, color="Green", alpha=0.6, label="Ratios", bins=100
    )
    s, loc, scale = lognorm.fit(ratios, loc=0)
    pdf = lognorm.pdf(binedges, s, loc, scale)
    twin3 = ax[ix, 3].twinx()
    twin3.set_ylim(0, 1.1 * np.max(pdf))
    twin3.plot(binedges, pdf, color="k", ls="--", label="Lognormal Fit")

    for a in [*ax[ix, :], twin2, twin3]:
        a.set_yticks([])

plt.tight_layout()
```



The form of these distributions is a reflection of the fact that geochemical data is at is core a measure of relative quantities of atoms. Quantities of atoms have discrete distributions (i.e. you can have precisely 0, 1 or 6.02 x 10^23 atoms, but 1.23 atoms is not a sensible state of affairs); if you were to count them in a shiny machine, the amount of atoms you might measure over a given period will have a Poisson distribution. If you measure two components, the probability density distribution of the ratio is well approximated by a lognormal distribution (note this doesn't consider inherent covariance):

```
from pyrolite.util.distributions import norm_to_lognorm
from pyrolite.util.plot.axes import share_axes, subaxes

# starting from a normal distribution, then creating similar non-normal distributions
mean, sd = 2.5, 1.5
logmu, logs = norm_to_lognorm(mean, sd)  # parameters for equival
normrv = norm(loc=mean, scale=sd)
lognormrv = lognorm(s=logs, scale=logmu)
poissonrv = poisson(mu=mean)
```

We can visualise the similarities and differences between these distributions:

```
fig, ax = plt.subplots(2, 3, figsize=(8, 4))
ax = ax.flat
for a in ax:
    a.subax = subaxes(a, side="bottom")

share_axes(ax[:3], which="x")
share_axes(ax[3:], which="x")
ax[0].set_xlim(-2, 10)
ax[3].set_xscale("log")
ax[3].set_xlim(0.1, 10)

for a in ax:
    a.axvline(0, color="k", lw=0.5, ls="--")

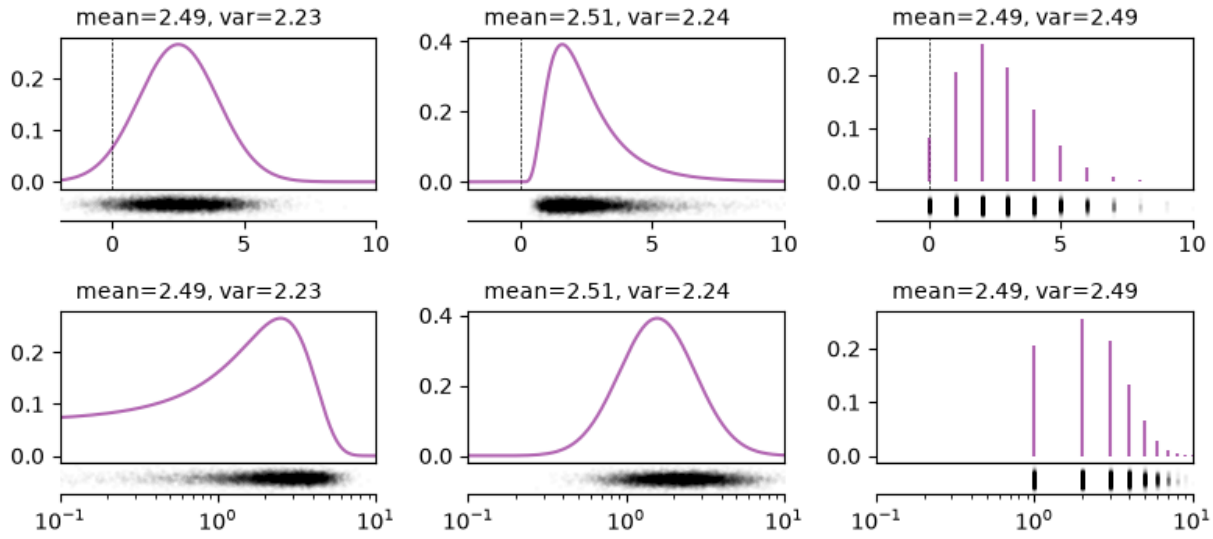
# xs at which to evaluate the pdfs
x = np.linspace(-5, 15.0, 1001)

for ix, dist in enumerate([normrv, lognormrv, poissonrv]):
    _xs = dist.rvs(size=10000)  # random sample
    _ys = -0.05 + np.random.randn(10000) / 100  # random offsets for visualisation
    for a in [ax[ix], ax[ix + 3]]:
        a.annotate(
            f"mean={np.mean(_xs):.2f}, var={np.var(_xs):.2f}",
            xy=(0.05, 1.05),
            ha="left",
            va="bottom",
            xycoords=a.transAxes,
        )
        a.subax.scatter(_xs, _ys, s=2, color="k", alpha=0.01)
        if dist != poissonrv:  # cont. distribution
            a.plot(x, dist.pdf(x), color="Purple", alpha=0.6, label="pdf")
        else:  # discrete distribution
            a.vlines(
                x[x >= 0],
                0,
                dist.pmf(x[x >= 0]),
                color="Purple",
                alpha=0.6,
                label="pmf",
            )
```

(continues on next page)

(continued from previous page)

```
fig.suptitle("Data Distributions: Normal, Lognormal, Poisson", y=1.1)
plt.tight_layout()
```



Accounting for these inherent features of geochemical data will allow you to accurately estimate means and variances, and from this enables the use of standardised statistical measures - as long as you're log-transforming your data. When performing multivariate analysis, use log-ratio transformations (including the additive logratio `alr()`, centred logratio `clr()` and isometric logratio `ilr()`). In this case, the logratio-mean is implemented for you:

```
import itertools

from pyrolite.comp.codata import logratiomean

fig, ax = plt.subplots(2, 2, figsize=(12, 12), subplot_kw=dict(projection="ternary"))
ax = ax.flat

for columns, a in zip(itertools.combinations(["A", "B", "C", "D"], 3), ax):
    columns = list(columns)

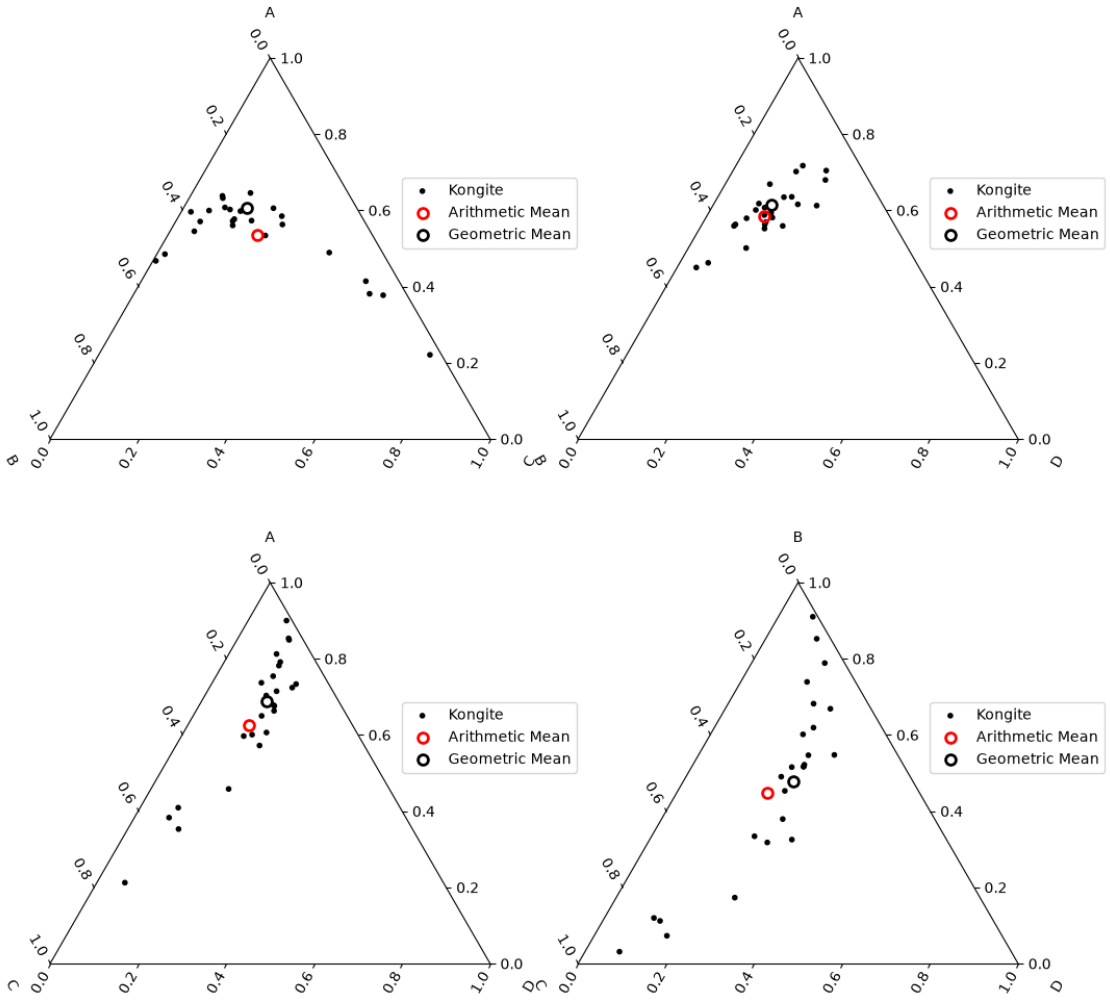
    df.loc[:, columns].pyroplot.scatter(
        ax=a, color="k", marker=".", label=df.attrs["name"], no_ticks=True
    )

    df.mean().loc[columns].pyroplot.scatter(
        ax=a,
        edgecolors="red",
        linewidths=2,
        c="none",
        s=50,
        label="Arithmetic Mean",
        no_ticks=True,
    )
```

(continues on next page)

(continued from previous page)

```
    logratiomean(df.loc[:, columns]).pyroplot.scatter(
        ax=a,
        edgecolors="k",
        linewidths=2,
        c="none",
        s=50,
        label="Geometric Mean",
        axlabels=True,
        no_ticks=True,
    )
    a.legend(loc=(0.8, 0.5))
```



See also

Examples:
Log Transforms, Logratio Means, Ternary Plots

Tutorials:
Ternary Density Plots, Making the Logo

Modules and Functions:
pyrolite.comp.codata, *renormalise()*

Total running time of the script: (0 minutes 2.032 seconds)

4.7 Utility Examples

pyrolite includes a range of utilities for everything from dealing with the web to plotting, synthetic data and machine learning. While most of these are used as part of the core functions of pyrolite, you may also find other uses for them, and this section provides some simple examples for some of these.

4.7.1 TAS Classifier

Some simple discrimination methods are implemented, including the Total Alkali-Silica (TAS) classification.

```
import matplotlib.pyplot as plt
import numpy as np

from pyrolite.util.classification import TAS
from pyrolite.util.synthetic import normal_frame
```

We'll first generate some synthetic data to play with:

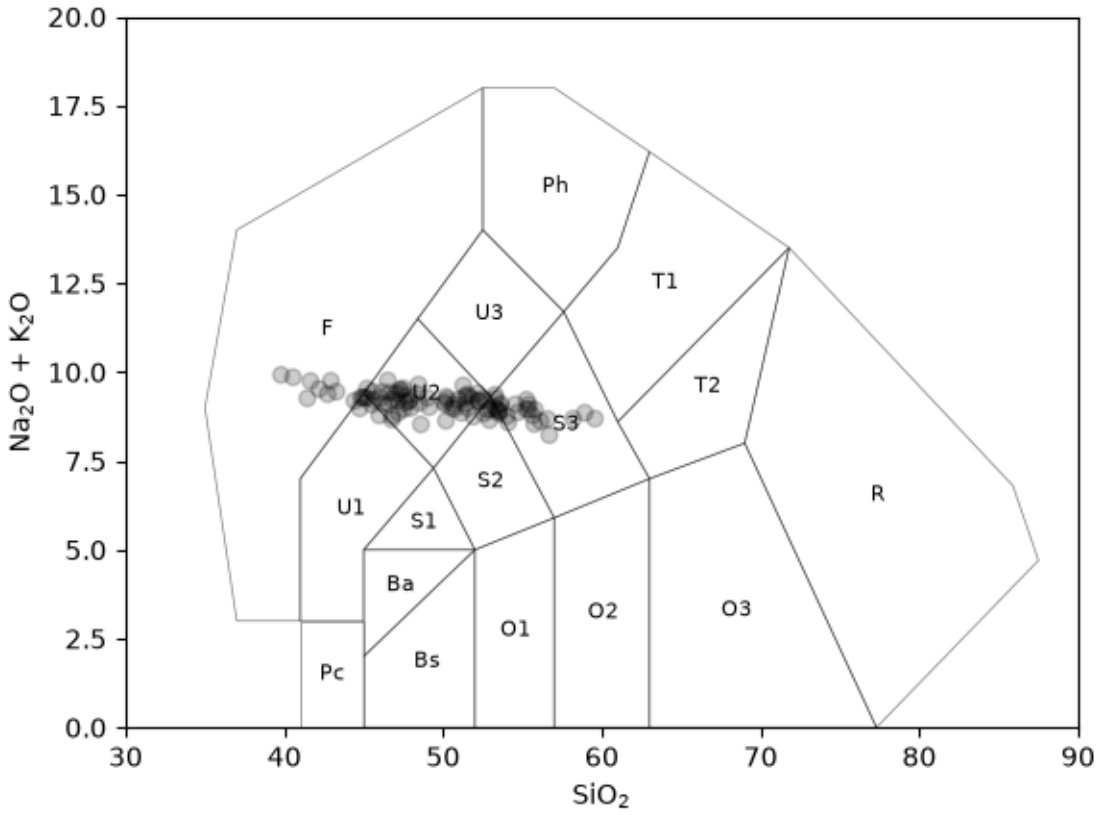
```
df = (
    normal_frame(
        columns=["SiO2", "Na2O", "K2O", "Al2O3"],
        mean=[0.5, 0.04, 0.05, 0.4],
        size=100,
        seed=49,
    )
    * 100
)

df.head(3)
```

We can visualise how this chemistry corresponds to the TAS diagram:

```
df["Na2O + K2O"] = df["Na2O"] + df["K2O"]
cm = TAS()

fig, ax = plt.subplots(1)
cm.add_to_axes(ax, alpha=0.5, linewidth=0.5, zorder=-1, add_labels=True)
df[["SiO2", "Na2O + K2O"]].pyroplot.scatter(ax=ax, c="k", alpha=0.2, axlabels=False)
plt.show()
```



We can now classify this data according to the fields of the TAS diagram, and add this as a column to the dataframe. Similarly, we can extract which rock names the TAS fields correspond to:

```
df["TAS"] = cm.predict(df)
df["Rocknames"] = df.TAS.apply(lambda x: cm.fields.get(x, {"name": None})["name"])
df["Rocknames"].sample(10)  # randomly check 10 sample rocknames
```

```
78    [Phonotephrite, Foid Monzodiorite]
2     [Phonotephrite, Foid Monzodiorite]
56    [Phonotephrite, Foid Monzodiorite]
41    [Phonotephrite, Foid Monzodiorite]
93    [Phonotephrite, Foid Monzodiorite]
60    [Phonotephrite, Foid Monzodiorite]
61    [Phonotephrite, Foid Monzodiorite]
47    [Phonotephrite, Foid Monzodiorite]
4         [Trachy-\nandesite, Monzonite]
12        [Trachy-\nandesite, Monzonite]
Name: Rocknames, dtype: object
```

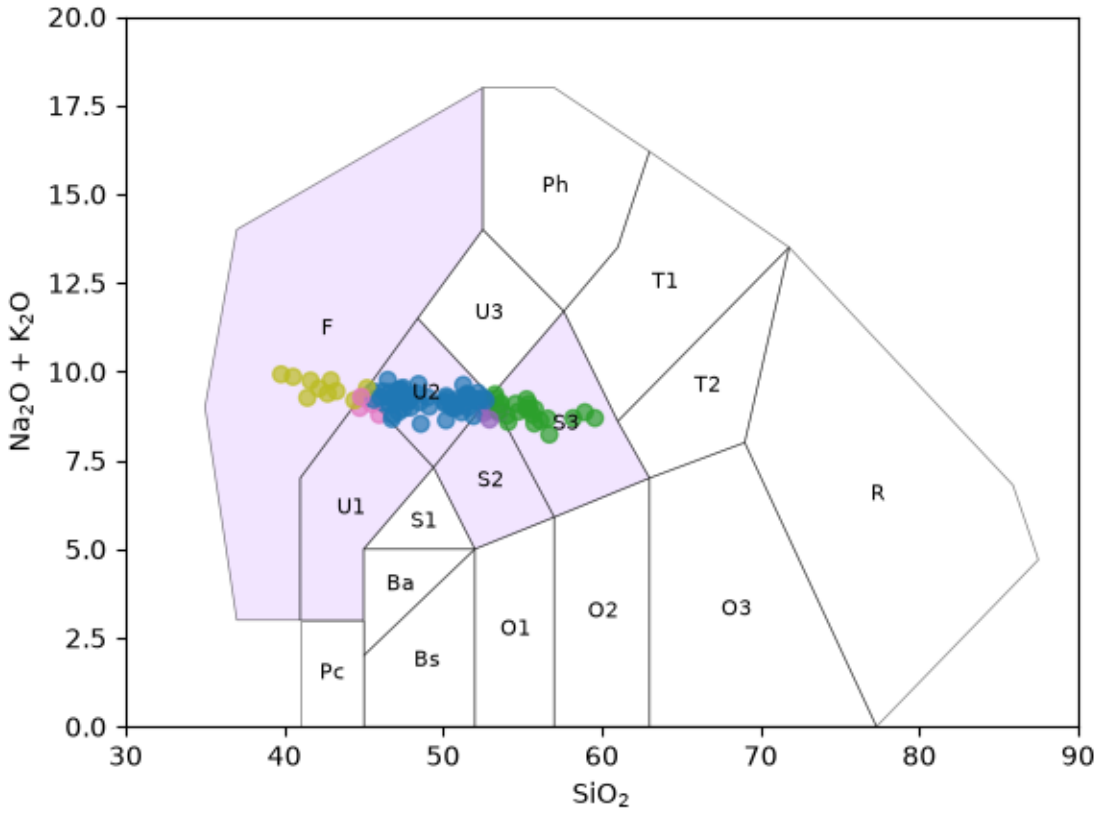
We could now take the TAS classes and use them to colorize our points for plotting on the TAS diagram, or more likely, on another plot. Here the relationship to the TAS diagram is illustrated, coloring also the populated fields:

```
fig, ax = plt.subplots(1)
```

(continues on next page)

(continued from previous page)

```
cm.add_to_axes(
    ax,
    alpha=0.5,
    linewidth=0.0,
    zorder=-2,
    add_labels=False,
    which_ids=np.unique(df["TAS"]),
    fill=True,
    facecolor=[0.9, 0.8, 1.0],
)
cm.add_to_axes(ax, alpha=0.5, linewidth=0.5, zorder=-1, add_labels=True)
df[["SiO2", "Na2O + K2O"]].pyroplot.scatter(
    ax=ax, c=df["TAS"], alpha=0.7, axlabels=False
)
```




```
<Axes: xlabel='SiO$_2$', ylabel='Na$_2$O + K$_2$O'>
```

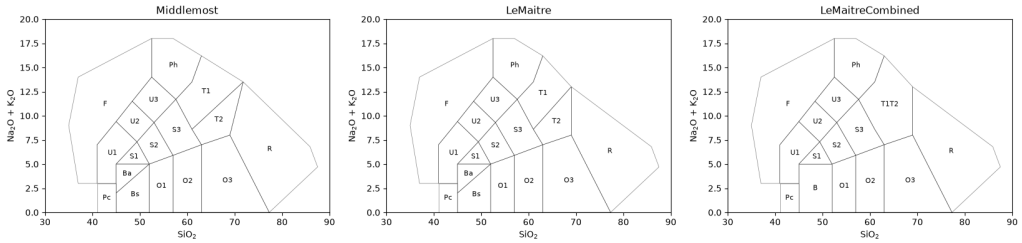
Variations of the Diagram

To use a different variation of the TAS diagram, you can pass the relevant keyword `which_model` allowing you to access the other available variants.

Currently, the Le Bas/Le Maitre alternative and a T1-T2 combined variant of it are available as alternatives to the

Middlemost version. Each of these can be used as a classifier model as for the default above.

```
fig, ax = plt.subplots(1, 3, figsize=(20, 4))
for a, model in zip(ax, ["Middlemost", "LeMaitre", "LeMaitreCombined"]):
    a.set_title(model if model is not None else "")
    cm = TAS(which_model=model)
    cm.add_to_axes(a, alpha=0.5, linewidth=0.5, add_labels=True)
plt.show()
```



References & Citation

For a few references on the TAS diagram as used here:

Middlemost, E. A. K. (1994).Naming materials in the magma/igneous rock system. Earth-Science Reviews, 37(3), 215–224. doi: 10.1016/0012-8252(94)90029-9.

Le Bas, M.J., Le Maitre, R.W., Woolley, A.R. (1992). The construction of the Total Alkali-Silica chemical classification of volcanic rocks. Mineralogy and Petrology 46, 1–22. doi: 110.1007/BF01160698.

Le Maitre, R.W. (2002). Igneous Rocks: A Classification and Glossary of Terms : Recommendations of International Union of Geological ciences Subcommission on the Systematics of Igneous Rocks. Cambridge University Press, 236pp. doi: 10.1017/CBO9780511535581.

Total running time of the script: (0 minutes 0.306 seconds)

4.7.2 Magnetite Classifier

Magnetite trace element discrimination diagrams for mineral deposit types (Dupuis and Beaudoin, 2011) are implemented as log-log polygon classifiers.

```
import matplotlib.pyplot as plt
import numpy as np

from pyrolite.util.classification import Magnetite
from pyrolite.util.synthetic import normal_frame
```

We'll first generate some synthetic magnetite compositions (wt%) to play with:

```
df = (
    normal_frame(
        columns=["Fe", "Ti", "V", "Ca", "Al", "Mn", "Ni", "Cr"],
        mean=[0.99, 0.002, 0.001, 0.0005, 0.002, 0.001, 0.0003, 0.0005],
        cov=np.eye(7) * 0.8,
        size=100,
        seed=49,
    )
    * 100
```

(continues on next page)

(continued from previous page)

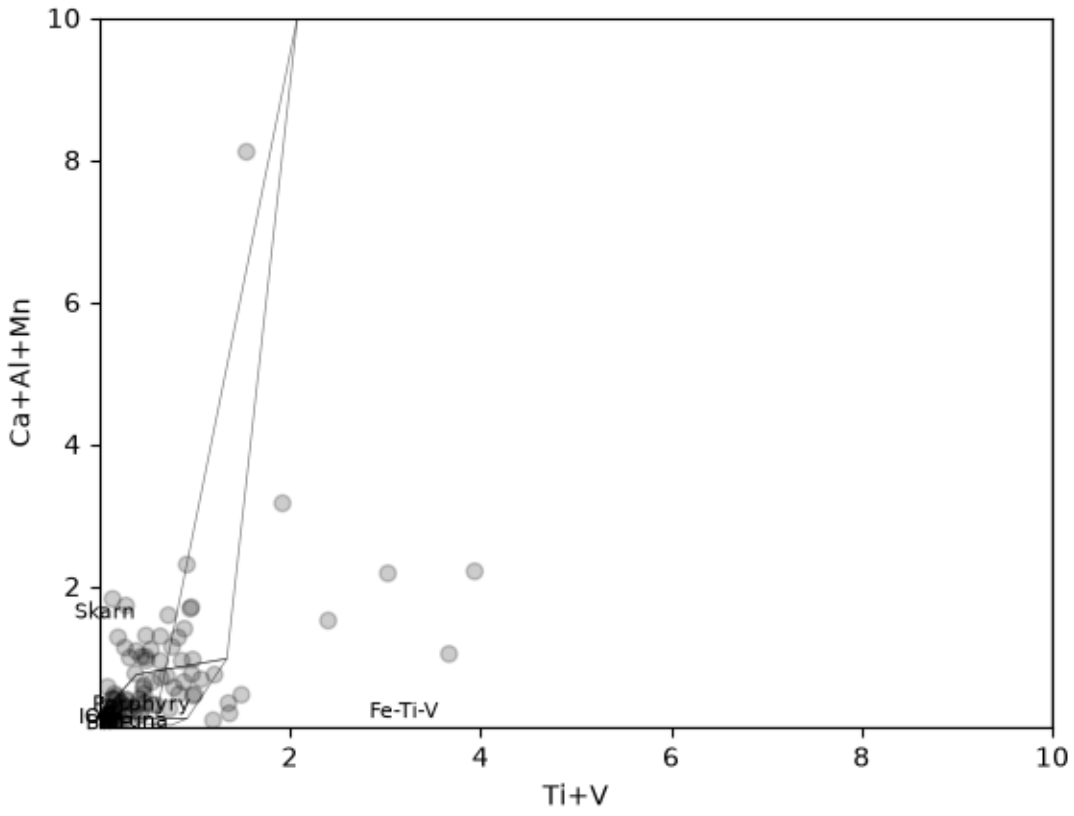
```
)

df.head(3)
```

We can visualise how this chemistry corresponds to the Ca+Al+Mn vs Ti+V diagram:

```
df["Ti+V"] = df["Ti"] + df["V"]
df["Ca+Al+Mn"] = df["Ca"] + df["Al"] + df["Mn"]
df["Ni/(Cr+Mn)"] = df["Ni"] / (df["Cr"] + df["Mn"])
cm = Magnetite()

fig, ax = plt.subplots(1)
cm.add_to_axes(ax, alpha=0.5, linewidth=0.5, zorder=-1, add_labels=True)
df[["Ti+V", "Ca+Al+Mn"]].pyroplot.scatter(ax=ax, c="k", alpha=0.2, axlabels=False)
plt.show()
```



We can now classify this data according to the fields of the diagram, and add this as a column to the dataframe:

```
df["Magnetite"] = cm.predict(df)
df["Magnetite"].sample(10)  # randomly check 10 sample classifications
```

```
56    Porphyry
82       Skarn
```

(continues on next page)

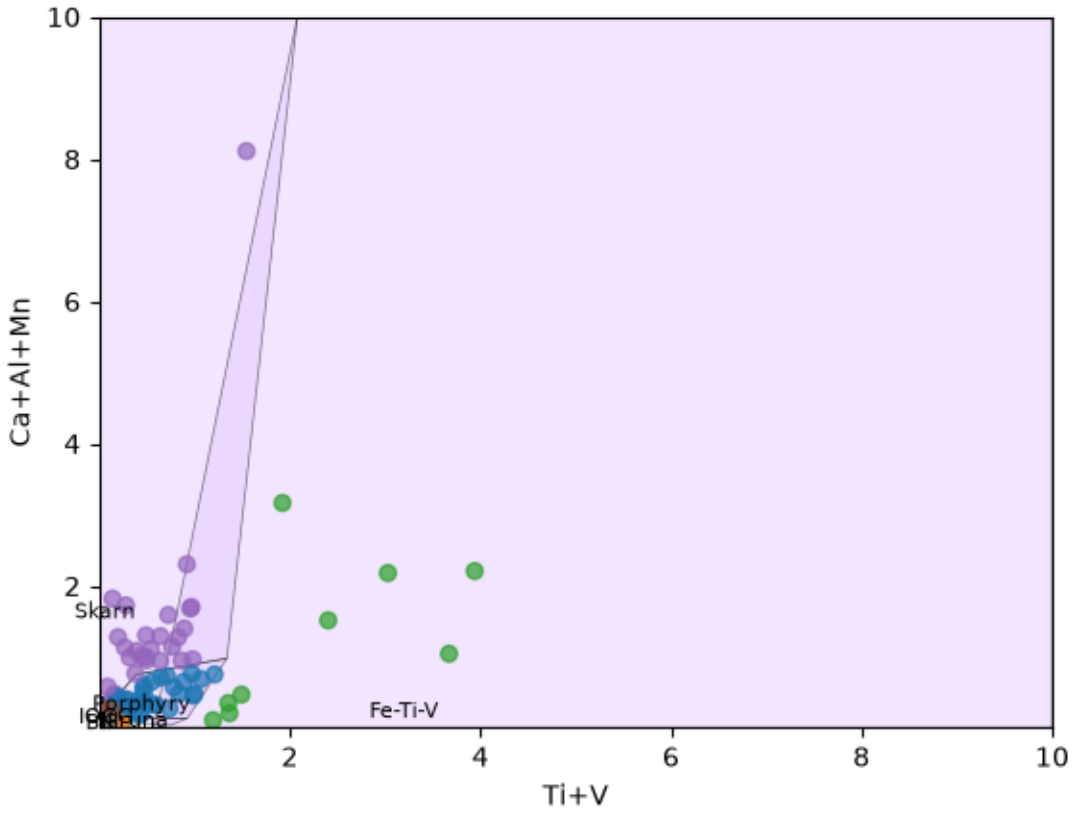
(continued from previous page)

```
46    Porphyry
39        IOCG
75        IOCG
77    Porphyry
43       Skarn
89       Skarn
78         BIF
20    Porphyry
Name: Magnetite, dtype: str
```

We could now take the classes and use them to colorize our points for plotting on the diagram, or more likely, on another plot. Here the relationship to the diagram is illustrated, coloring also the populated fields:

```
fig, ax = plt.subplots(1)

cm.add_to_axes(
    ax,
    alpha=0.5,
    linewidth=0.0,
    zorder=-2,
    add_labels=False,
    which_ids=np.unique(df["Magnetite"]),
    fill=True,
    facecolor=[0.9, 0.8, 1.0],
)
cm.add_to_axes(ax, alpha=0.5, linewidth=0.5, zorder=-1, add_labels=True)
df[["Ti+V", "Ca+Al+Mn"]].pyroplot.scatter(
    ax=ax, c=df["Magnetite"], alpha=0.7, axlabels=False
)
```

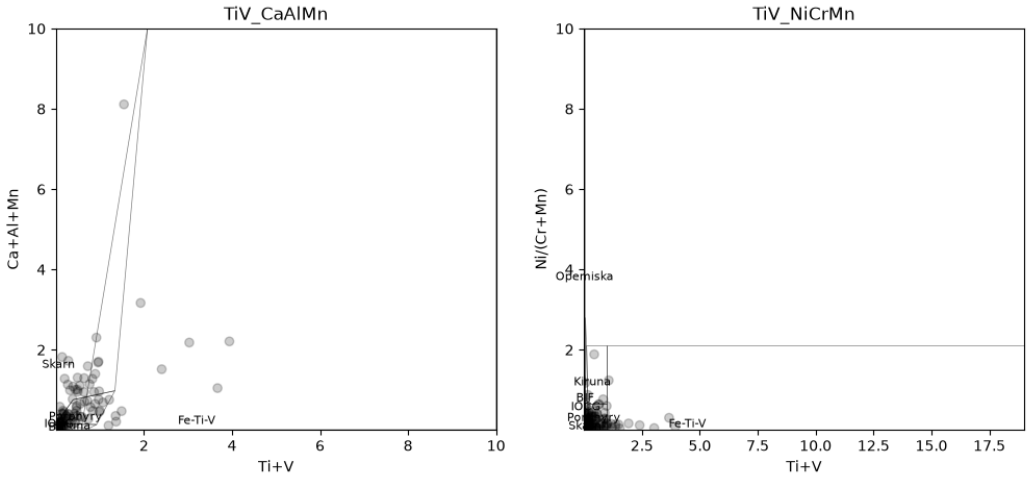



```
<Axes: xlabel='Ti+V', ylabel='Ca+Al+Mn'>
```

Variations of the Diagram

To use the other diagram, pass the relevant keyword `which_model`. Currently, the Ca+Al+Mn vs Ti+V (`"TiV_CaAlMn"`, default) and Ni/(Cr+Mn) vs Ti+V (`"TiV_NiCrMn"`) diagrams are available. Each of these can be used as a classifier model as for the default above.

```
models = {"TiV_CaAlMn": "Ca+Al+Mn", "TiV_NiCrMn": "Ni/(Cr+Mn)"}
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
for a, (model, y) in zip(ax, models.items()):
    a.set_title(model)
    cm = Magnetite(which_model=model)
    cm.add_to_axes(a, alpha=0.5, linewidth=0.5, add_labels=True)
    df[["Ti+V", y]].pyroplot.scatter(ax=a, c="k", alpha=0.2, axlabels=False)
plt.show()
```



References & Citation

Dupuis, C., Beaudoin, G. (2011). Discriminant diagrams for iron oxide trace element fingerprinting of mineral deposit types. Mineralium Deposita 46, 319-335. doi: 10.1007/s00126-011-0334-y.

Nadoll, P. (2011). Geochemistry of magnetite from hydrothermal ore deposits and host rocks – Case studies from the Proterozoic Belt Supergroup, Cu-Mo-porphyry + skarn and Climax-Mo deposits in the western United States. hdl: hdl.handle.net/2292/7512.

Nadoll, P., Angerer, T., Mauk, J. L., French, D., & Walshe, J. (2014). The chemistry of hydrothermal magnetite: A review. Ore Geology Reviews, 61, 1–32. doi: 10.1016/j.oregeorev.2013.12.013.

Total running time of the script: (0 minutes 0.183 seconds)

4.7.3 Geological Timescale

pyrolite includes a simple geological timescale, based on a recent verion of the International Chronostratigraphic Chart[1]. The *Timescale* class can be used to look up names for specific geological ages, to look up times for known geological age names and to access a reference table for all of these.

First we'll create a timescale:

```
from pyrolite.util.time import Timescale

ts = Timescale()
```

From this we can look up the names of ages (in million years, or Ma):

```
ts.named_age(1212.1)
```

```
'Ectasian'
```

As geological age names are hierarchical, the name you give an age depends on what level you're looking at. By default, the timescale will return the most specific non-null level. The levels accessible within the timescale are listed as an attribute:

```
ts.levels
```

[1] Cohen, K.M., Finney, S.C., Gibbard, P.L., Fan, J.-X., 2013. The ICS International Chronostratigraphic Chart. Episodes 36, 199–204.

```
['Supereon', 'Eon', 'Era', 'Period', 'Superepoch', 'Epoch', 'Age']
```

These can be used to refine the output names to your desired level of specificity (noting that for some ages, the levels which are accessible can differ; see the chart):

```
ts.named_age(1212.1, level="Epoch")
```

```
'Ectasian'
```

The timescale can also do the inverse for you, and return the timing information for a given named age:

```
ts.text2age("Holocene")
```

```
(0.0117, 0.0)
```

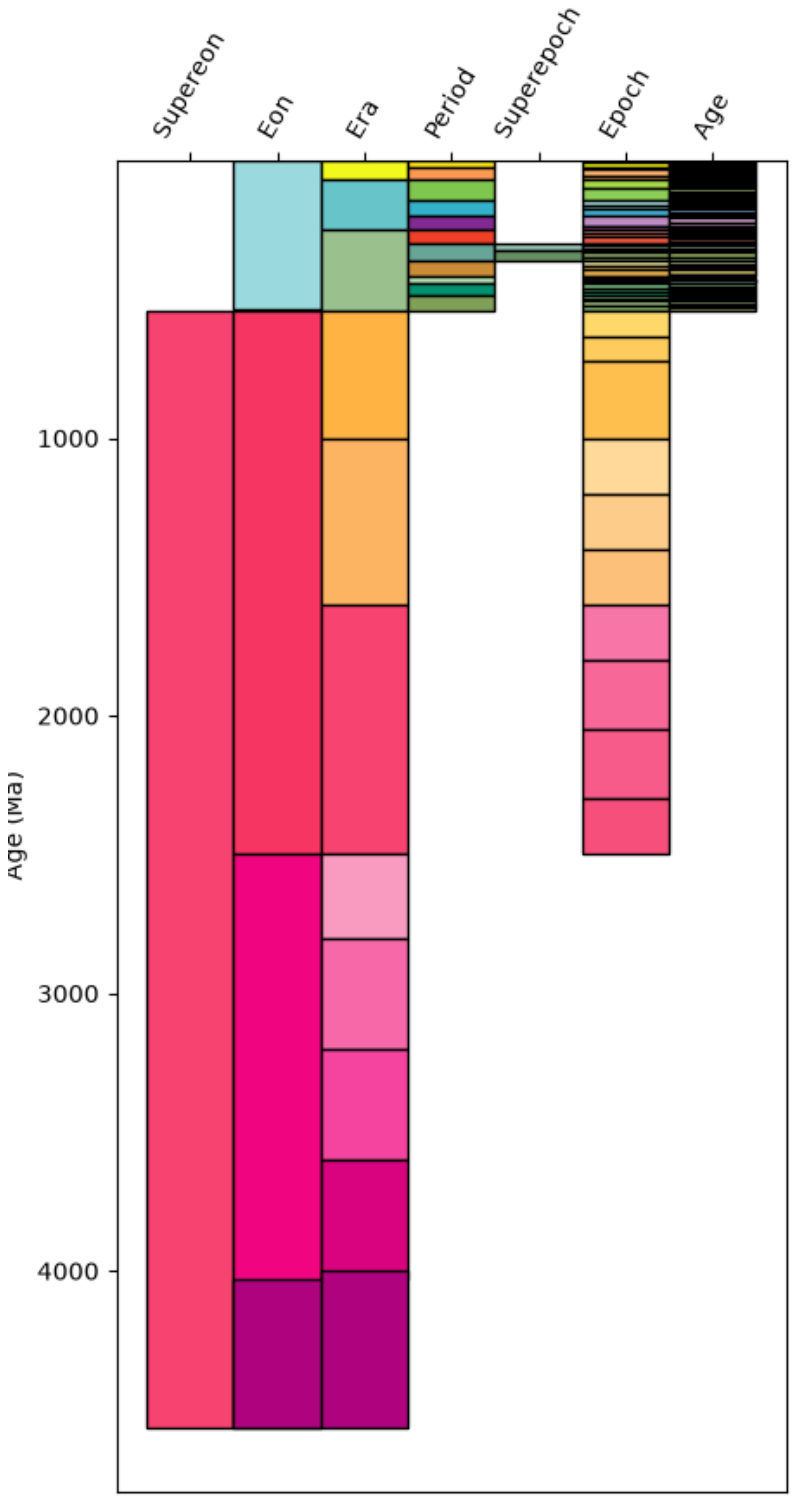
We can use this to create a simple template to visualise the geological timescale:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, figsize=(5, 10))

for ix, level in enumerate(ts.levels):
    ldf = ts.data.loc[ts.data.Level == level, :]
    for pix, period in ldf.iterrows():
        ax.bar(
            ix,
            period.Start - period.End,
            facecolor=period.Color,
            bottom=period.End,
            width=1,
            edgecolor="k",
        )

ax.set_xticks(range(len(ts.levels)))
ax.set_xticklabels(ts.levels, rotation=60)
ax.xaxis.set_ticks_position("top")
ax.set_ylabel("Age (Ma)")
ax.invert_yaxis()
```


Supereon
Eon
Era
Period
Superepoch
Epoch
Age
1000
2000
3000
4000
Age (Ma)

This doesn't quite look like the geological timescale you may be used to. We can improve on the output somewhat with a bit of customisation for the positioning. Notably, this is less readable, but produces something closer to what we're after. Some of this may soon be integrated as a *Timescale* method, if there's interest.

```
import numpy as np
from matplotlib.patches import Rectangle

# first let's set up some x-limits for the different timescale levels
xlims = {
    "Eon": (0, 1),
    "Era": (1, 2),
    "Period": (2, 3),
    "Superepoch": (3, 4),
    "Epoch": (3, 5),
    "Age": (5, 7),
}


fig, ax = plt.subplots(1, figsize=(4, 10))

for ix, level in enumerate(ts.levels[::-1]):
    if level in xlims:
        ldf = ts.data.loc[ts.data.Level == level, :]
        for pix, period in ldf.iterrows():
            left, right = xlims[level]
            time = np.mean(ts.text2age(period.Name))
            if ix != len(ts.levels) - 1:
                general_bound = None
                _ix = ix
                while general_bound is None:
                    try:
                        _ix += 1
                        bound_level = ts.levels[::-1][_ix]
                        general_bound = ts.named_age(time, level=bound_level)
                    except IndexError:
                        break
                if bound_level in xlims:
                    _l, _r = xlims[bound_level]
                    left = max(left, _r)

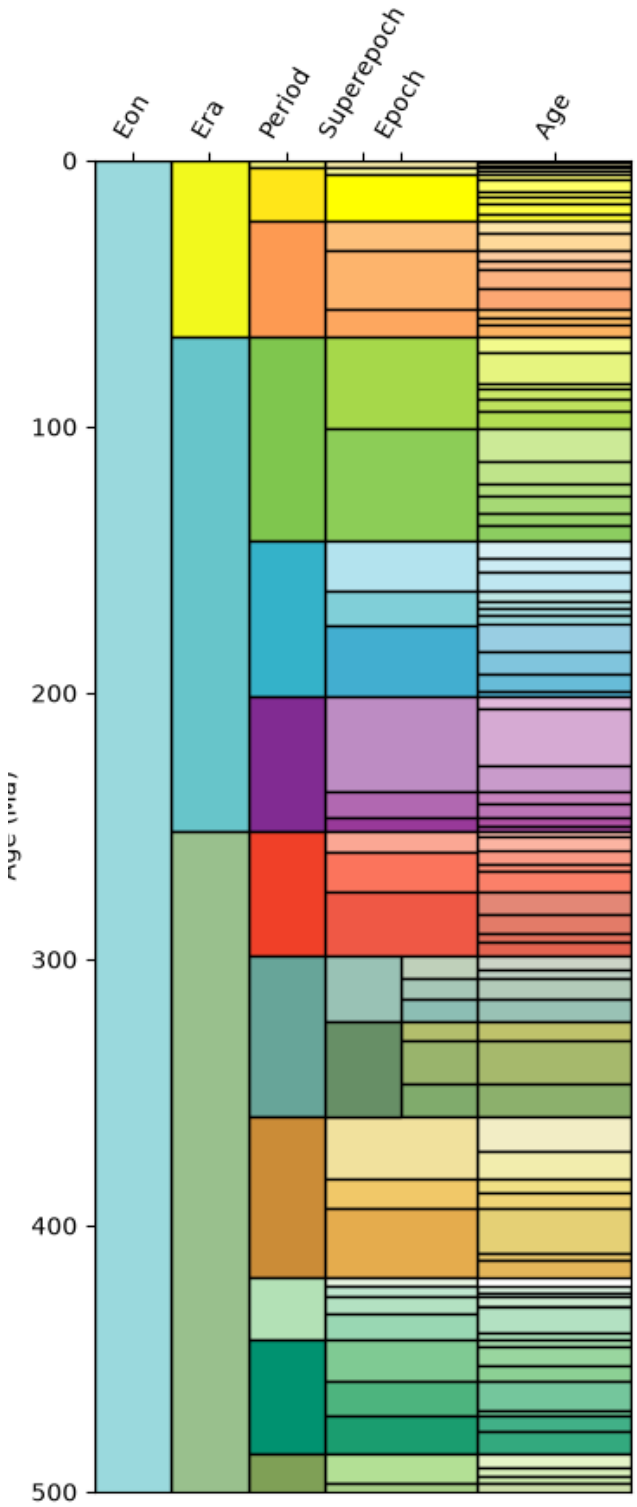
            rect = Rectangle(
                (left, period.End),
                right - left,
                period.Start - period.End,
                facecolor=period.Color,
                edgecolor="k",
            )
            ax.add_artist(rect)

ax.set_xticks([np.mean(xlims[lvl]) for lvl in xlims])
ax.set_xticklabels(xlims.keys(), rotation=60)
ax.xaxis.set_ticks_position("top")
ax.set_xlim(0, 7)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("Age (Ma)")
ax.set_ylim(500, 0)
```


Eon
Era
Period
Superepoch
Epoch
Age
0
100
200
300
400
500

```
(500.0, 0.0)
```

Total running time of the script: (0 minutes 0.784 seconds)

4.8 Tutorials

This page is home to longer examples which incorporate multiple components of `pyrolite` or provide examples of how to integrate these into your own workflows.

> **Note**
>
> This page is a work in progress. Feel free to request tutorials or examples with a feature request.

4.8.1 Creating Plot Templates/Classifier Models

`pyrolite` provides a system for creating and using plot templates/classifier models based on a series of polygons in variable space (e.g., the TAS diagram). A variety of diagram templates/ classifiers are available, but you can also create your own.

In this tutorial, we'll go through the process of creating a diagram template from scratch, as a demonstration of how you might create your own for your use - or to later contribute to the collection in pyrolite.

The basis for most diagrams and classifiers is the class *PolygonClassifier*; the docstring-based help text is a good place to start to understand what we'll need to put it together:

```
from pyrolite.util.classification import PolygonClassifier

help(PolygonClassifier)
```

```
Help on class PolygonClassifier in module pyrolite.util.classification:

class PolygonClassifier(builtins.object)
 |  PolygonClassifier(
 |      name: str | None = None,
 |      axes: dict | None = None,
 |      fields: dict | None = None,
 |      scale: float = 1.0,
 |      transform: collections.abc.Callable | str | None = None,
 |      mode: str | None = None,
 |      **kwargs
 |  )
 |
 |  A classifier model built form a series of polygons defining specific classes.
 |
 |  Parameters
 |  -----------
 |  name : str
 |      A name for the classifier model.
 |  axes : dict
 |      Mapping from plot axes to variables to be used for labels.
 |  fields : dict
 |      Dictionary describing indiviudal polygons, with identifiers as keys and
```

(continues on next page)

(continued from previous page)

```
|      dictionaries containing 'name' and 'fields' items.
|  scale : float
|      Default maximum scale for the axes. Typically 100 (wt%) or 1 (fractional).
|  xlim : tuple
|      Default x-limits for this classifier for plotting.
|  ylim : tuple
|      Default y-limits for this classifier for plotting.
|
|  Methods defined here:
|
|  __init__(
|      self,
|      name: str | None = None,
|      axes: dict | None = None,
|      fields: dict | None = None,
|      scale: float = 1.0,
|      transform: collections.abc.Callable | str | None = None,
|      mode: str | None = None,
|      **kwargs
|  )
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  add_to_axes(
|      self,
|      ax: matplotlib.axes._axes.Axes | None = None,
|      fill: bool = False,
|      axes_scale: float = 1.0,
|      add_labels: bool = False,
|      which_labels: str = 'ID',
|      which_ids: list[str] | None = None,
|      **kwargs
|  ) -> matplotlib.axes._axes.Axes
|      Add the fields from the classifier to an axis.
|
|      Parameters
|      ----------
|      ax : matplotlib.axes.Axes
|          Axis to add the polygons to.
|      fill : bool
|          Whether to fill the polygons.
|      axes_scale : float
|          Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
|      add_labels : bool
|          Whether to add labels for the polygons.
|      which_labels : str
|          Which data to use for field labels - field 'name' or 'ID'.
|      which_ids : list
|          List of field IDs corresponding to the polygons to add to the axes object.
|          (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields).
|          An empty list corresponds to plotting all the polygons.
|
|      Returns
```

(continues on next page)

(continued from previous page)

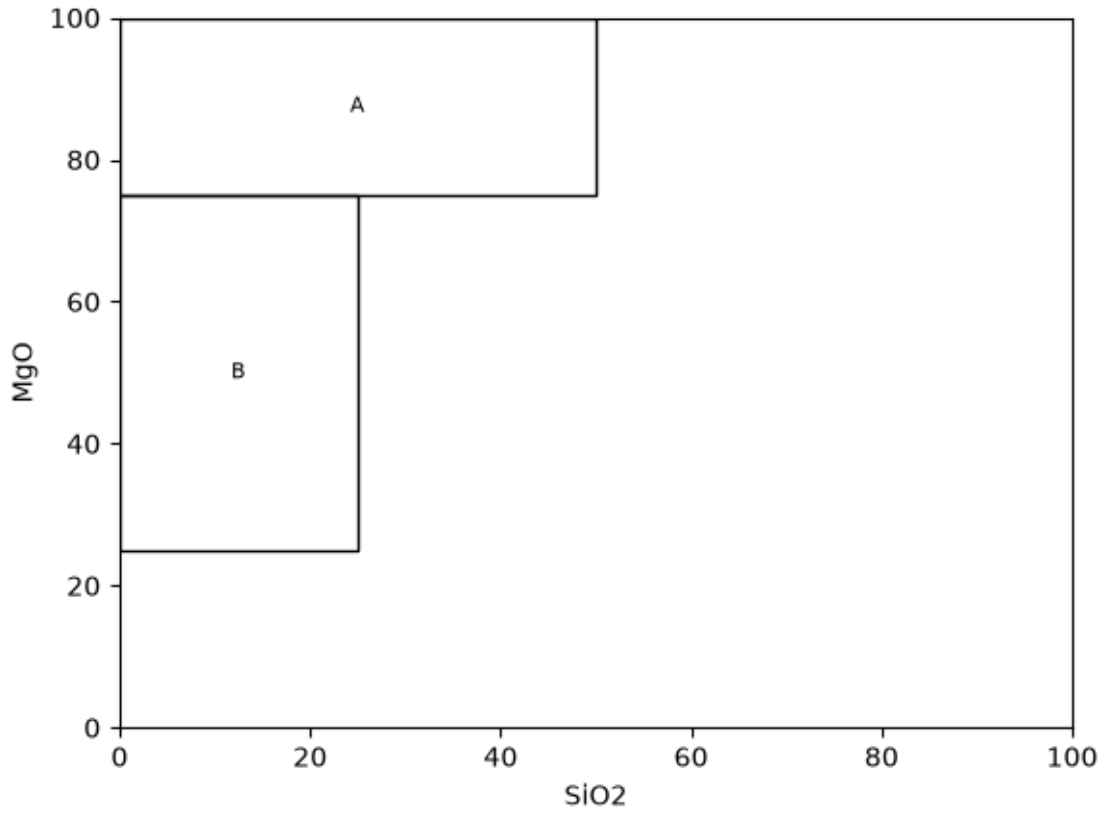
```
|      --------
|      ax : matplotlib.axes.Axes
|
|  predict(
|      self,
|      X: numpy.ndarray[tuple[int, int], numpy.dtype[numpy.number]] | pandas.DataFrame,
|      data_scale=None
|  ) -> pandas.Series
|      Predict the classification of samples using the polygon-based classifier.
|
|      Parameters
|      -----------
|      X : numpy.ndarray | pandas.DataFrame
|          Data to classify.
|      data_scale : float
|          Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).
|
|      Returns
|      -------
|      pandas.Series
|          Series containing classifer predictions. If a dataframe was input,
|          it inherit the index.
|
|  ----------------------------------------------------------------------
|  Readonly properties defined here:
|
|  axis_components
|      Get the axis components used by the classifier.
|
|      Returns
|      -------
|      tuple
|          Ordered names for axes used by the classifier.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables
|
|  __weakref__
|      list of weak references to the object
```

The key things you'll need to construct a classifer are:

1. a name
2. a specification of what the axes correspond to,
3. and a dictionary of fields (dictionaries containing a 'name' and coordinates defining the polygon).

We can also optionally specify the x and y limits, which are specific to plotting. Here we'll put together a simple classifier model with just two fields, and add this to a `matplotlib` axis. You can optionally specify names/labels for each field, here we opt to just use some basic IDs (A and B), so these are what will be added to the plot:

```
clf = PolygonClassifier(
    name="DemoClassifier",
    axes={"x": "SiO2", "y": "MgO"},
    fields={
        "A": {
            "poly": [[0, 75], [0, 100], [50, 100], [50, 75]],
        },
        "B": {
            "poly": [[0, 25], [0, 75], [25, 75], [25, 25]],
        },
    },
    xlim=(0, 100),
    ylim=(0, 100),
)
ax = clf.add_to_axes(add_labels=True)
ax.figure
```




```
<Figure size 640x480 with 1 Axes>
```

While we're individually passing each of these arguments to *PolygonClassifier*, we can also pass a dictionary of keyword arguments:

```
cfg = dict(
```

(continues on next page)

(continued from previous page)

```
    name="DemoClassifier",
    axes={"x": "SiO2", "y": "MgO"},
    fields={
        "A": {
            "poly": [[0, 75], [0, 100], [50, 100], [50, 75]],
        },
        "B": {
            "poly": [[0, 25], [0, 75], [25, 75], [25, 25]],
        },
    },
    xlim=(0, 100),
    ylim=(0, 100),
)

clf = PolygonClassifier(**cfg)
```

Each of the built-in models are saved as JSON files, and loaded in a manner as above; we can replicate that here - saving our configuration to JSON then loading it up again. We'll use a temporary directory here, but you can save it wherever you like (note the `pyrolite` templates live under */data/models* in the repository); once you've got a template working how you'd like, consider submitting it!

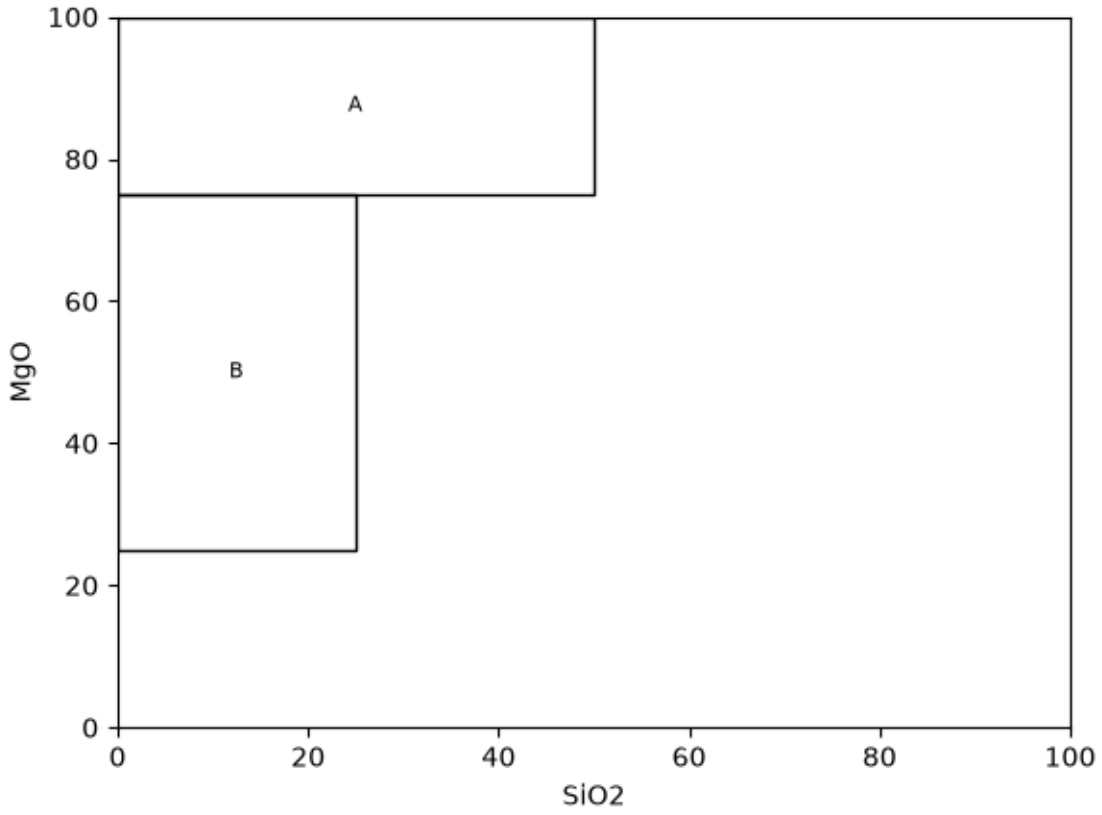
```
import json

from pyrolite.util.general import temp_path

tmp = temp_path()
with open(tmp / "demo_classifier.json", "w") as f:
    f.write(json.dumps(cfg))


with open(tmp / "demo_classifier.json", "r") as f:
    clf = PolygonClassifier(**json.load(f))

clf.add_to_axes(add_labels=True).figure
```

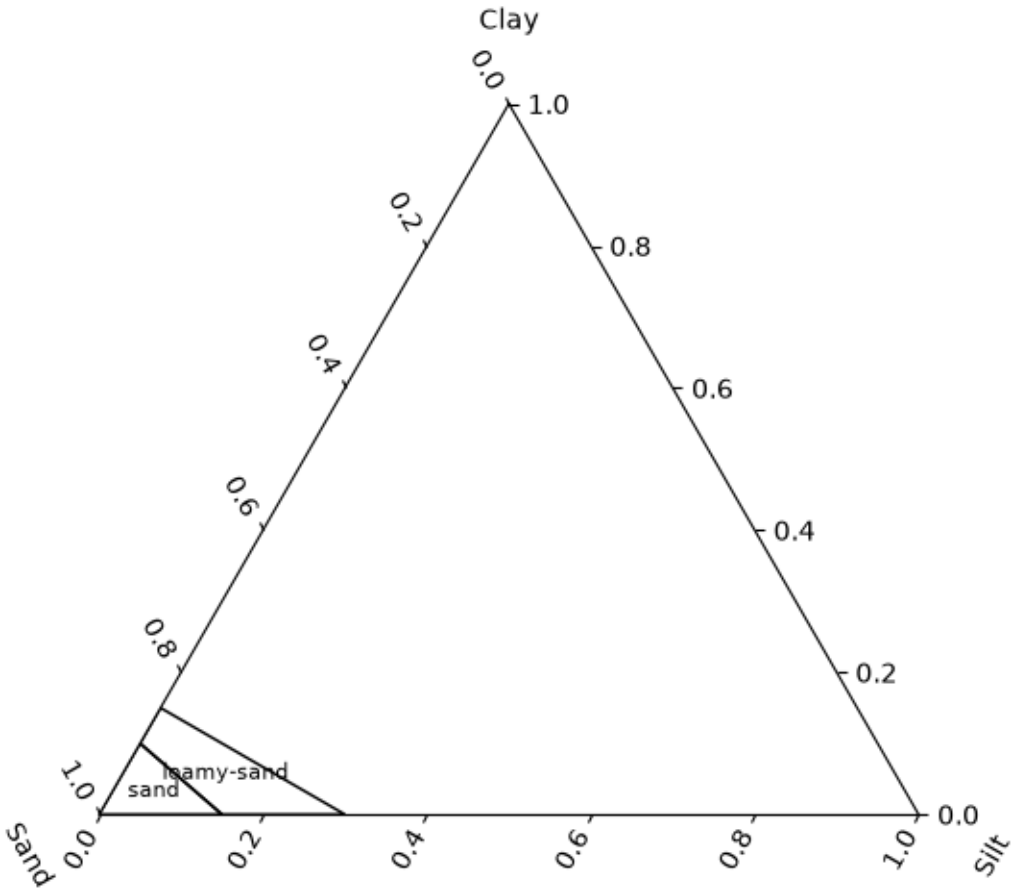



```
<Figure size 640x480 with 1 Axes>
```

Ternary Templates

While it's slightly more work, you can also generate ternary templates using a very simliar pattern to the bivariate ones above. The principal differences are that you'll need to specify three axes (t, l, r), specify a 'ternary' transform, and have coordinates for polygons in the ternary space - each with three values. For example, here are two fields from the UDSA soil texture triangle:

```
cfg = {
    "axes": {"t": "Clay", "l": "Sand", "r": "Silt"},
    "transform": "ternary",
    "fields": {
        "sand": {"name": "Sand", "poly": [[0, 100, 0], [10, 90, 0], [0, 85, 15]]},
        "loamy-sand": {
            "name": "Loamy Sand",
            "poly": [[10, 90, 0], [0, 85, 15], [0, 70, 30], [15, 85, 0]],
        },
    },
}
PolygonClassifier(**cfg).add_to_axes(add_labels=True).figure
```




```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:56: UserWarning:␣
↪Non-positive entries found. Closure operation assumes all positive entries.
  warnings.warn(

<Figure size 640x480 with 1 Axes>
```

Total running time of the script: (0 minutes 0.251 seconds)

4.8.2 Formatting and Cleaning Up Plots

> **Note**
>
> This tutorial is a work in progress and will be gradually updated.

In this tutorial we will illustrate some straightfoward formatting for your plots which will allow for greater customisation as needed. As `pyrolite` heavily uses and exposes the API of `matplotlib` for the visualisation components (and also `mpltern` for ternary diagrams), you should also check out their documentation pages for more in-depth guides, examples and API documentation.

First let's pull in a simple dataset to use throughout these examples:

```
from pyrolite.util.synthetic import normal_frame
```

(continues on next page)

(continued from previous page)

```
df = normal_frame(columns=["SiO2", "CaO", "MgO", "Al2O3", "TiO2", "27Al", "d11B"])
```

Basic Figure and Axes Settings

`matplotlib` makes it relatively straightfoward to customise most settings for your figures and axes. These settings can be defined at creation (e.g. in a call to `subplots()`), or they can be defined after you've created an axis (with the methods `ax.set_<parameter>()`). For example:

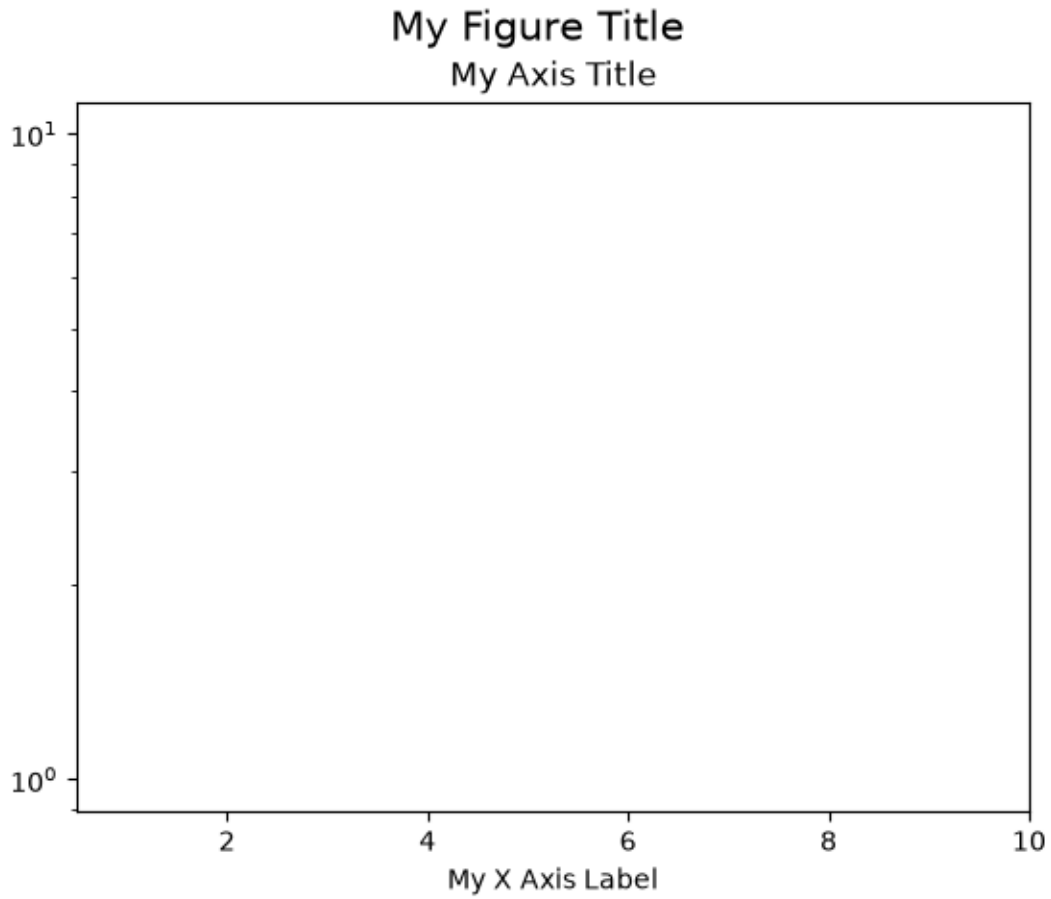
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1)

ax.set_xlabel("My X Axis Label")
ax.set_title("My Axis Title", fontsize=12)
ax.set_yscale("log")
ax.set_xlim((0.5, 10))

fig.suptitle("My Figure Title", fontsize=15)

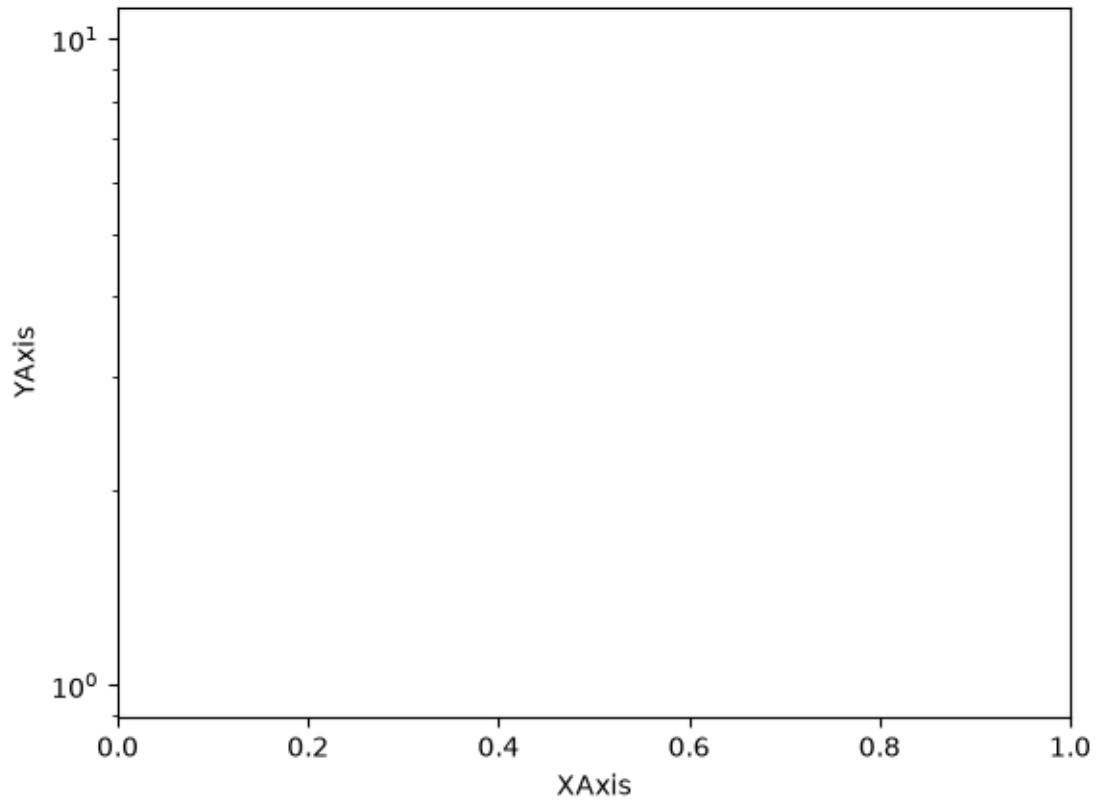
plt.show()
```



You can use a single method to set most of these things: `set()`. For example:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1)
ax.set(yscale="log", xlim=(0, 1), ylabel="YAxis", xlabel="XAxis")
plt.show()
```



Labels and Text

`matplotlib` enables you to use TEX within all text elements, including labels and annotations. This can be leveraged for more complex formatting, incorporating math and symbols into your plots. Check out the mod:*matplotlib* tutorial, and for more on working with text generally in `matplotlib`, check out the relevant tutorials gallery.

The ability to use TeX syntax in `matplotlib` text objects can also be used for typsetting, like for subscripts and superscripts. This is particularly relevant for geochemical oxides labels (e.g. Al2O3, which would ideally be rendered as Al_2O_3) and isotopes (e.g. d11B, which should be $\delta^{11}B$). At the moment, pyrolite won't do this for you, so you may want to adjust the labelling after you've made them. For example:

```
import matplotlib.pyplot as plt

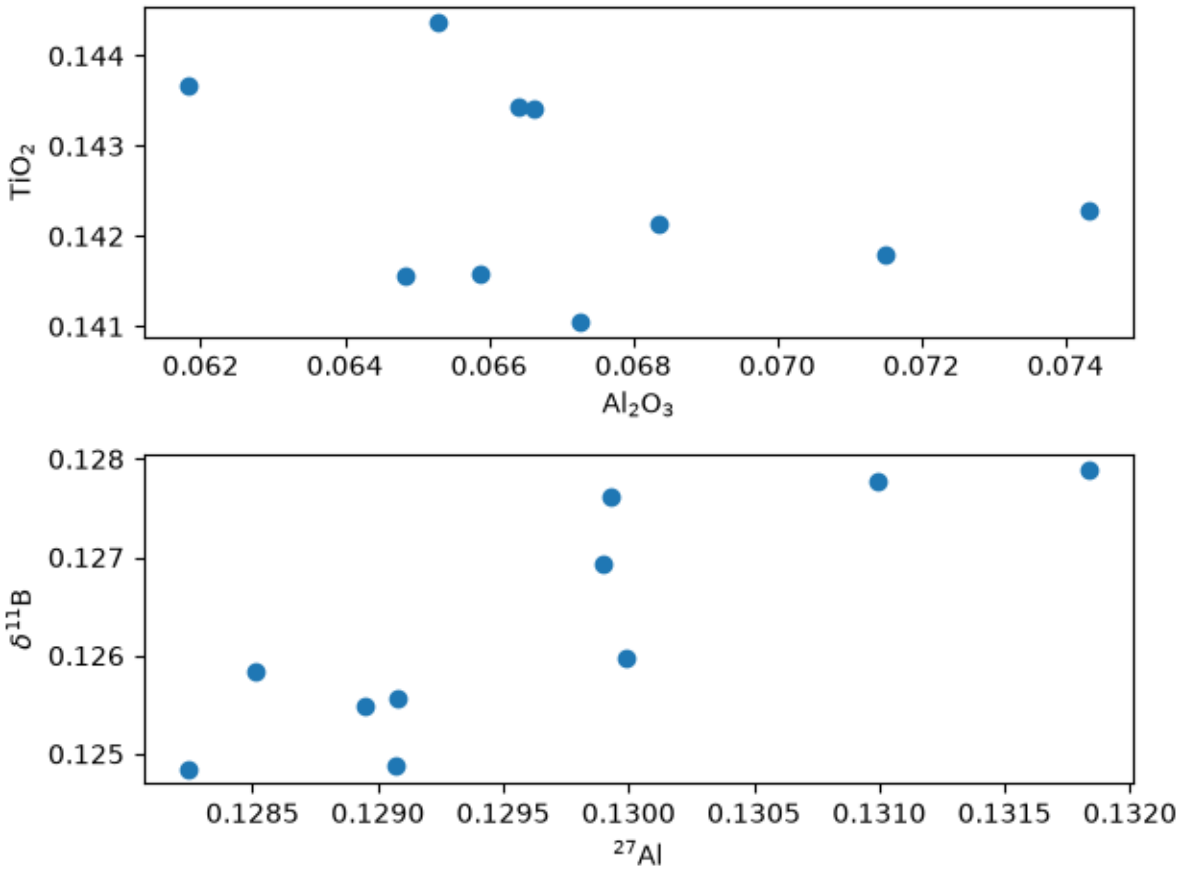
fig, ax = plt.subplots(2, 1)
df[["Al2O3", "TiO2"]].pyroplot.scatter(ax=ax[0])
ax[0].set_xlabel("Al$_2$O$_3$")
ax[0].set_ylabel("TiO$_2$")
```

(continues on next page)

(continued from previous page)

```
df[["27Al", "d11B"]].pyroplot.scatter(ax=ax[1])
ax[1].set_xlabel("$^{27}$Al")
ax[1].set_ylabel(r"$\delta^{11}$B")

plt.tight_layout()  # rearrange the plots to fit nicely together
plt.show()
```

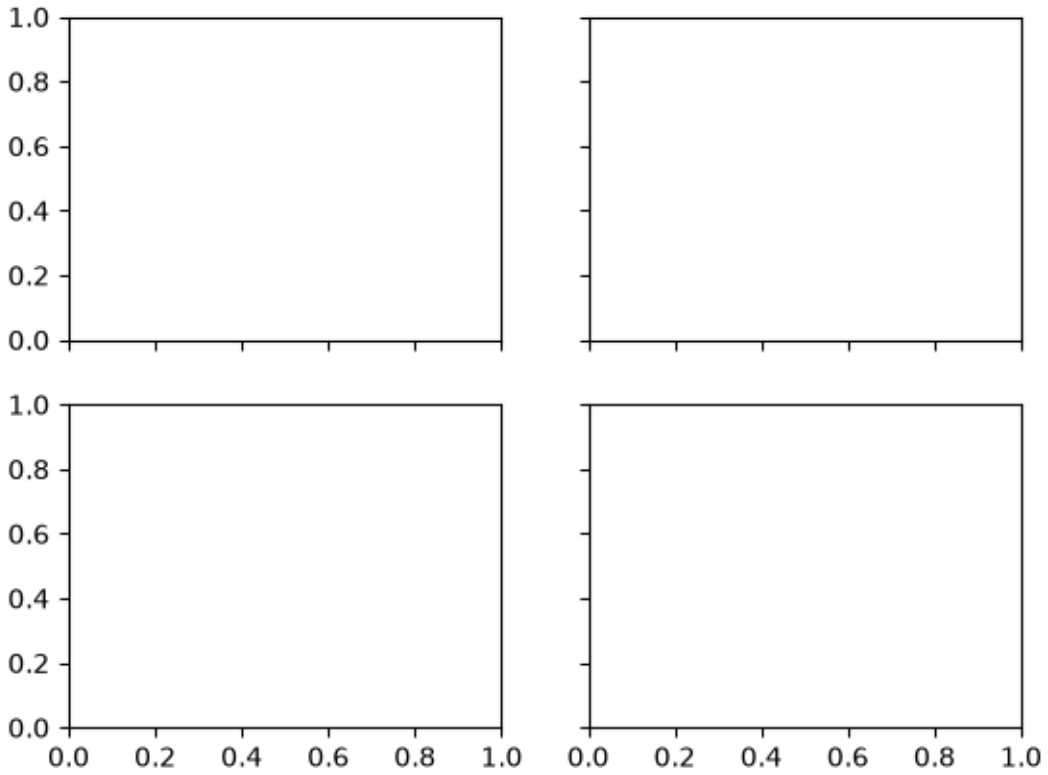


Sharing Axes

If you're building figures which have variables which are re-used, you'll typically want to 'share' them between your axes. The `matplotlib.pyplot` API makes this easy for when you want to share among *all* the axes as your create them:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)
plt.show()
```



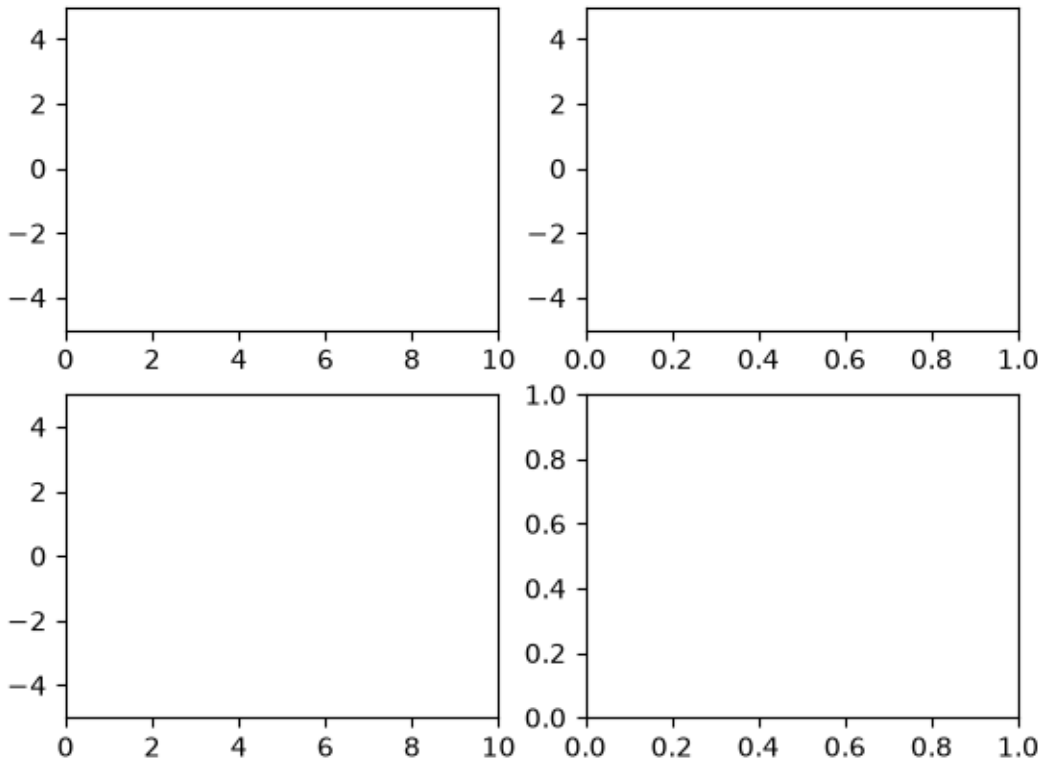
However, if you want to share axes in a way which is less standard, it can be difficult to set up using this function. `pyrolite` has a utility function which can be used to share axes after they're created in slightly more arbitrary ways. For example, imagine we wanted to share the first and third x-axes, and the first three y-axes, you could use:

```
import matplotlib.pyplot as plt

from pyrolite.util.plot.axes import share_axes

fig, ax = plt.subplots(2, 2)
ax = ax.flat  # turn the (2,2) array of axes into one flat axes with shape (4,)
share_axes([ax[0], ax[2]], which="x")  # share x-axes for 0, 2
share_axes(ax[0:3], which="y")  # share y-axes for 0, 1, 2

ax[0].set_xlim((0, 10))
ax[1].set_ylim((-5, 5))
plt.show()
```

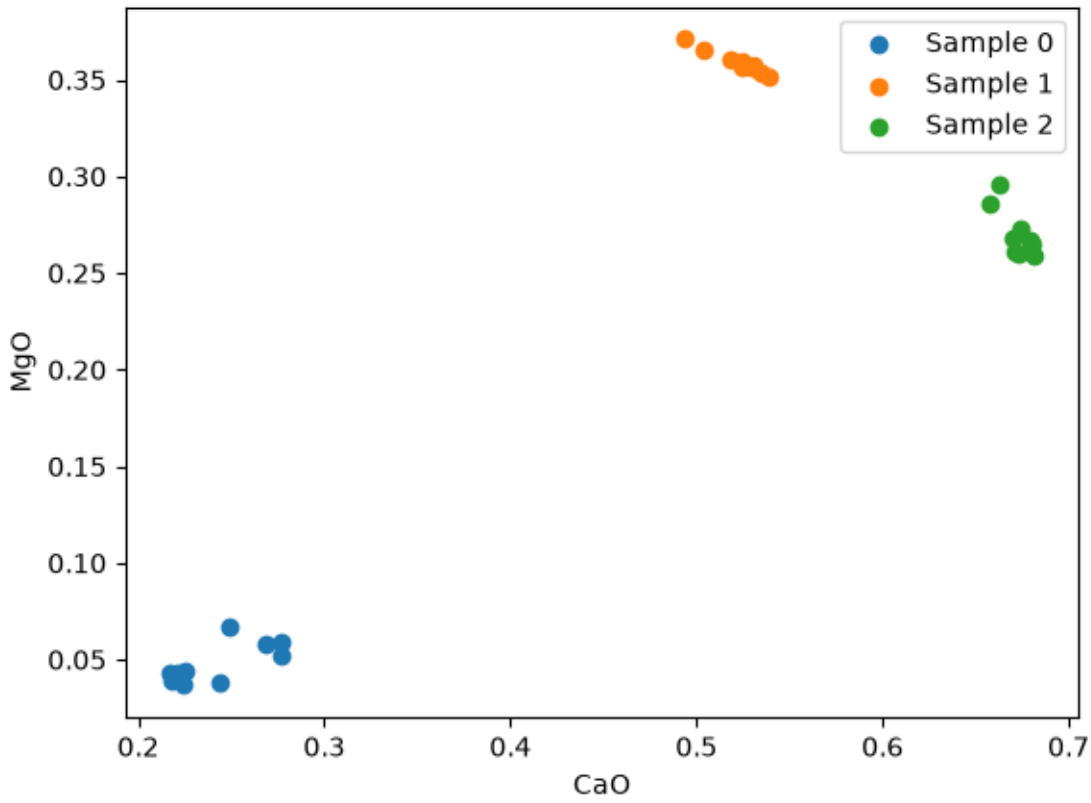


Legends

While it's simple to set up basic legends in `maplotlib` (see the docs for `matplotlib.axes.Axes.legend()`), often you'll want to customise your legends to fit nicely within your figures. Here we'll create a few synthetic datasets, add them to a figure and create the default legend:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1)
for i in range(3):
    sample_data = normal_frame(columns=["CaO", "MgO", "FeO"])  # a new random sample
    sample_data[["CaO", "MgO"]].pyroplot.scatter(ax=ax, label=f"Sample {i:d}")
ax.legend()
plt.show()
```



On many of the `pyrolite` examples, you'll find legends formatted along the lines of the following to clean them up a little (these are the default styles):

```
ax.legend(
    facecolor=None,  # have a transparent legend background
    frameon=False,  # remove the legend frame
    bbox_to_anchor=(1, 1),  # anchor legend's corner to the axes' top-right
    loc="upper left",  # use the upper left corner for the anchor
)
plt.show()
```

Check out the `matplotlib` legend guide for more.

Ternary Plots

The ternary plots in `pyrolite` are generated using `mpltern`, and while the syntax is very similar to the `matplotlib` API, as we have three axes to deal with sometimes things are little different. Here we demonstrate how to complete some common tasks, but you should check out the `mpltern` documentation if you want to dig deeper into customising your ternary diagrams (e.g. see the example gallery), which these examples were developed from.

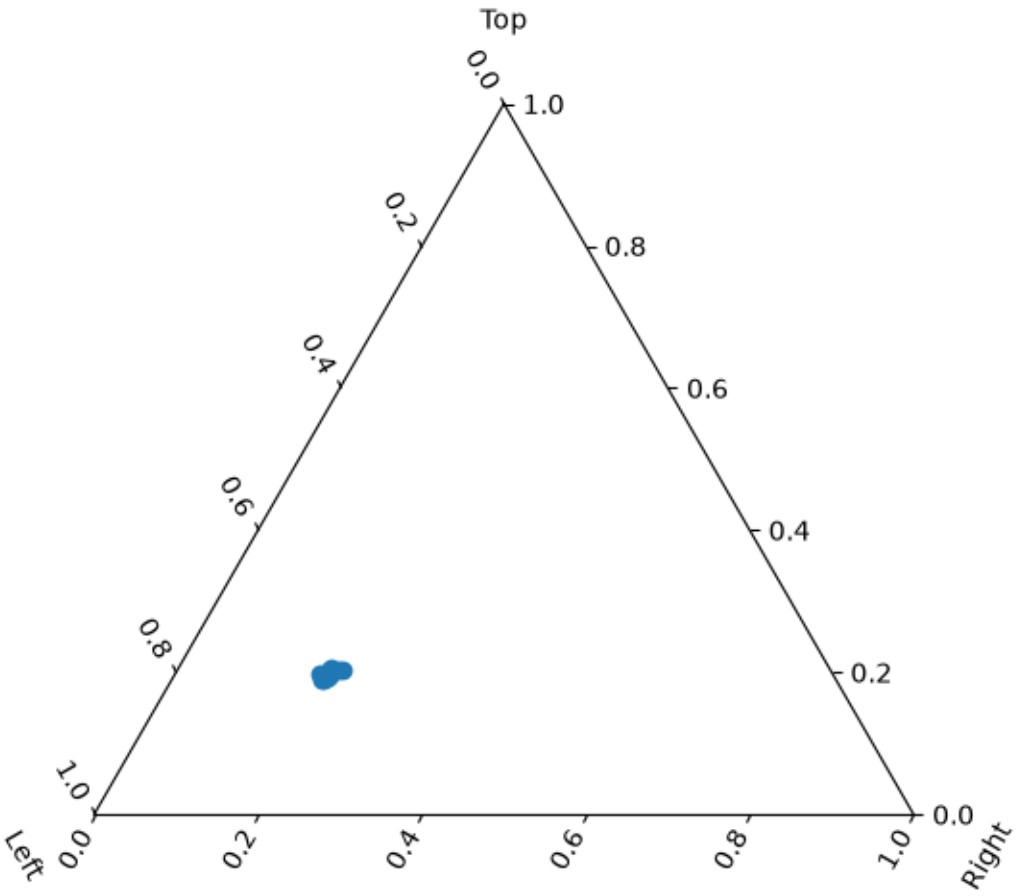
One of the key things to note in `mpltern` is that you have *top*, *left* and *right* axes.

Ternary Plot Axes Labels

Labelling ternary axes is done similarly to in `matplotlib`, but using the axes prefixes *t*, *l* and *r* for top, left and right axes, respectively:

```
import matplotlib.pyplot as plt

ax = df[["CaO", "MgO", "Al2O3"]].pyroplot.scatter()
ax.set_tlabel("Top")
ax.set_llabel("Left")
ax.set_rlabel("Right")
plt.show()
```

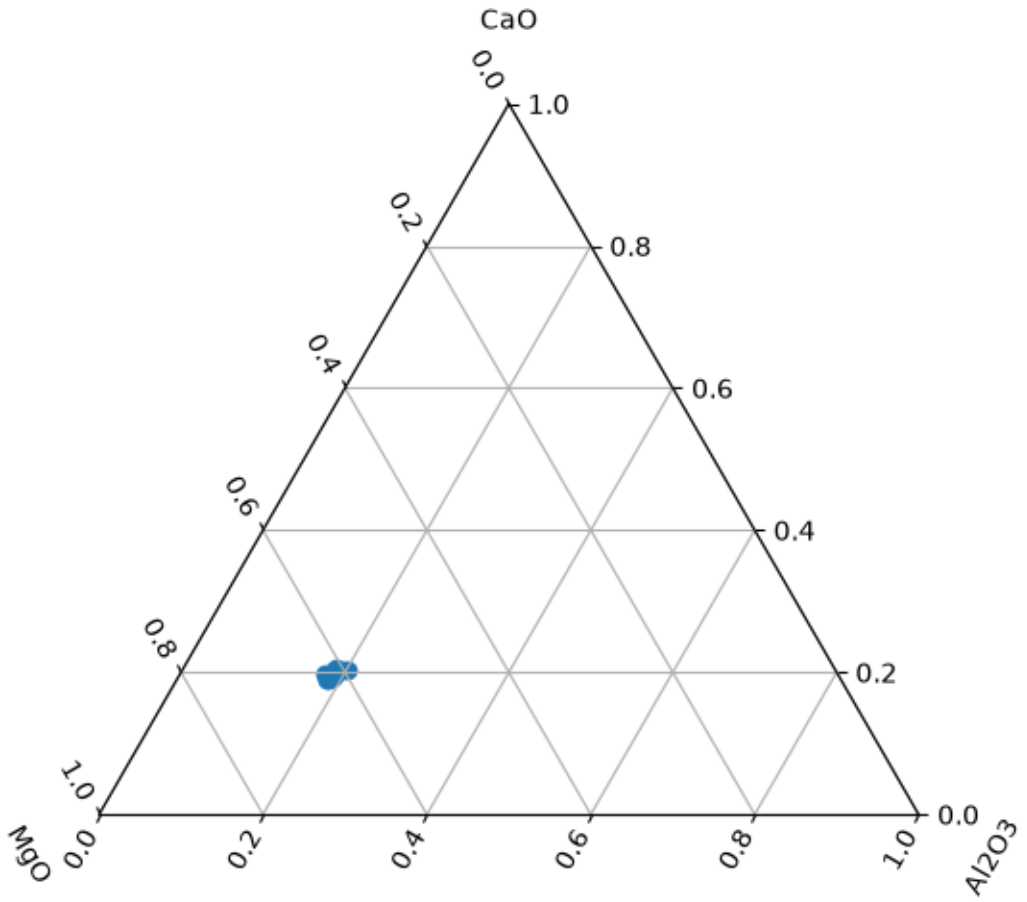


Ternary Plot Grids

To add a simple grid to your ternary plot, you can use `grid()`:

```
import matplotlib.pyplot as plt

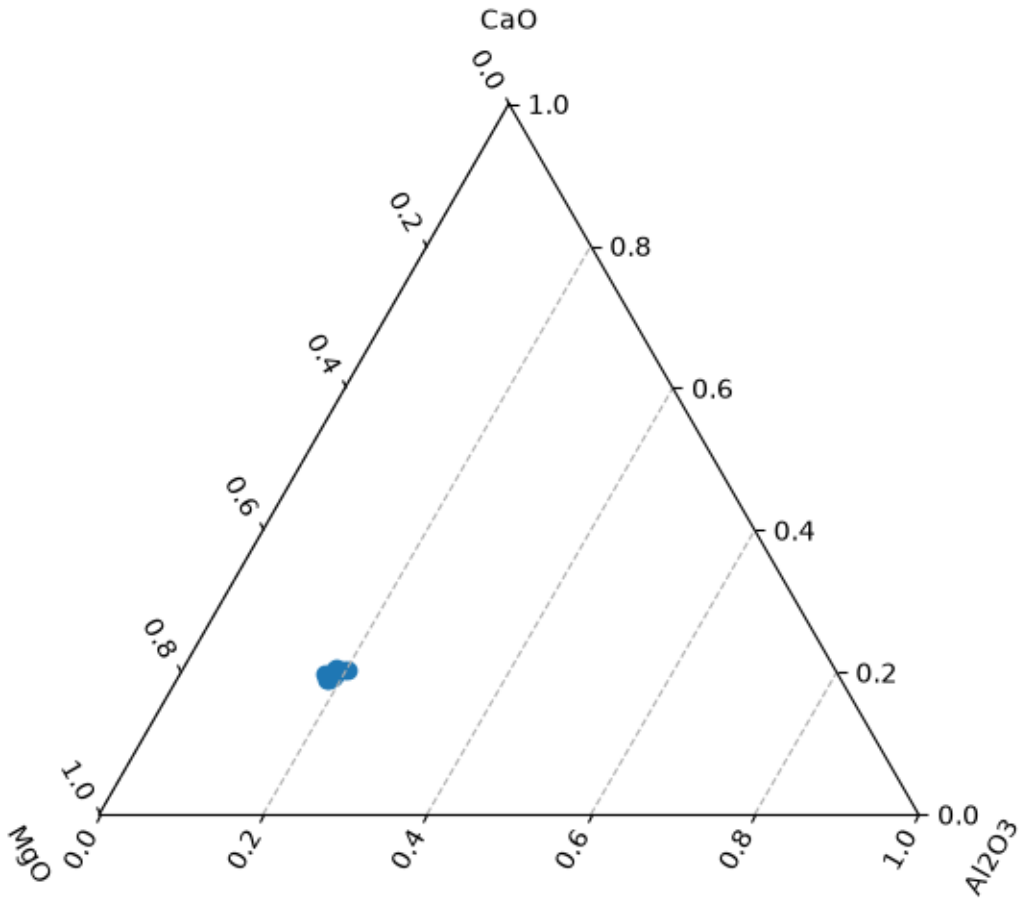
ax = df[["CaO", "MgO", "Al2O3"]].pyroplot.scatter()
ax.grid()
plt.show()
```



With this method, you can also specify an *axis*, *which* tickmarks you want to use for the grid ('major', 'minor' or 'both') and a *linestyle*:

```
import matplotlib.pyplot as plt

ax = df[["CaO", "MgO", "Al2O3"]].pyroplot.scatter()
ax.grid(axis="r", which="both", linestyle="--")
plt.show()
```



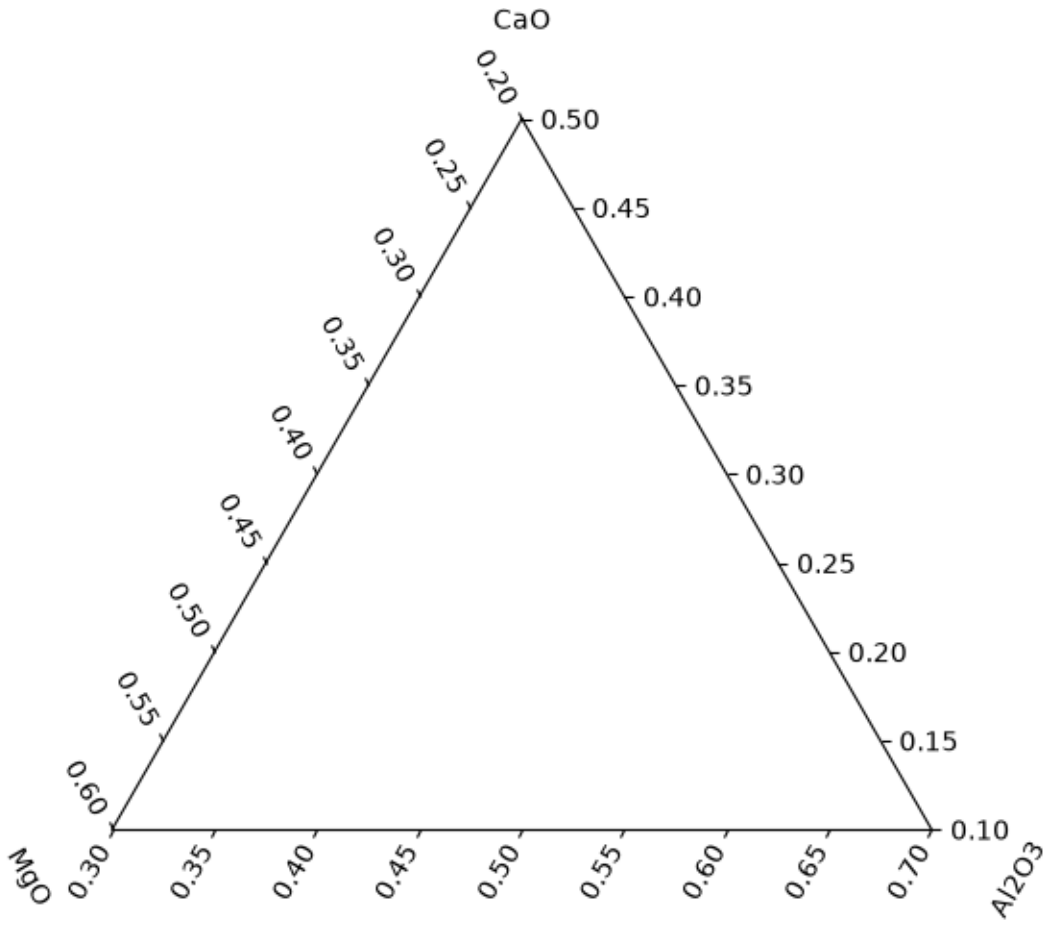
Ternary Plot Limits

To focus on a specific area, you can reset the limits of your ternary axes with `set_ternary_lim()`.

Also check out the `mpltern` inset axes example if you're after ways to focus on specific regions.

```
import matplotlib.pyplot as plt

ax = df[["CaO", "MgO", "Al2O3"]].pyroplot.scatter()
ax.set_ternary_lim(
    0.1,
    0.5,
    0.2,
    0.6,
    0.3,
    0.7,  # tmin  # tmax  # lmin  # lmax  # rmin  # rmax
)
plt.show()
```



Total running time of the script: (0 minutes 1.227 seconds)

4.8.3 Making the Logo

Having some funky ellipses in a simplex inspired some interest when I put the logo together for pyrolite, so I put together a cleaned-up example of how you can create these kinds of plots for your own data. These examples illustrate different methods to show distribution of (homogeneous, or near so) compositional data for exploratory analysis.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from pyrolite.comp.codata import *
from pyrolite.util.plot.helpers import plot_pca_vectors, plot_stdev_ellipses
from pyrolite.util.skl.transform import ILRTransform
from pyrolite.util.synthetic import random_composition

np.random.seed(82)

# ignore sphinx_gallery warnings
import warnings

warnings.filterwarnings("ignore", "Matplotlib is currently using agg")
```

First we choose some colors, create some log-distributed synthetic data. Here I've generated a synthetic dataset with

four samples having means equidistant from the log-space centre and with varying covariance. This should illustrate the spatial warping of the simplex nicely. Additionally, I chose a log-transform here to go from and to compositional space (`ILRTransform`, which uses the isometric log-ratio function `ilr()`). Choosing another transform will change the distortion observed in the simplex slightly. This synthetic dataset is added into a `DataFrame` for convenient access to plotting functions via the pandas API defined in *pyrolite.plot.pyroplot*.

```
t10b3 = [  # tableau 10 colorblind safe colors, a selection of 4
    (0, 107, 164),
    (171, 171, 171),
    (89, 89, 89),
    (95, 158, 209),
]
t10b3 = [(r / 255.0, g / 255.0, b / 255.0) for r, g, b in t10b3]
```

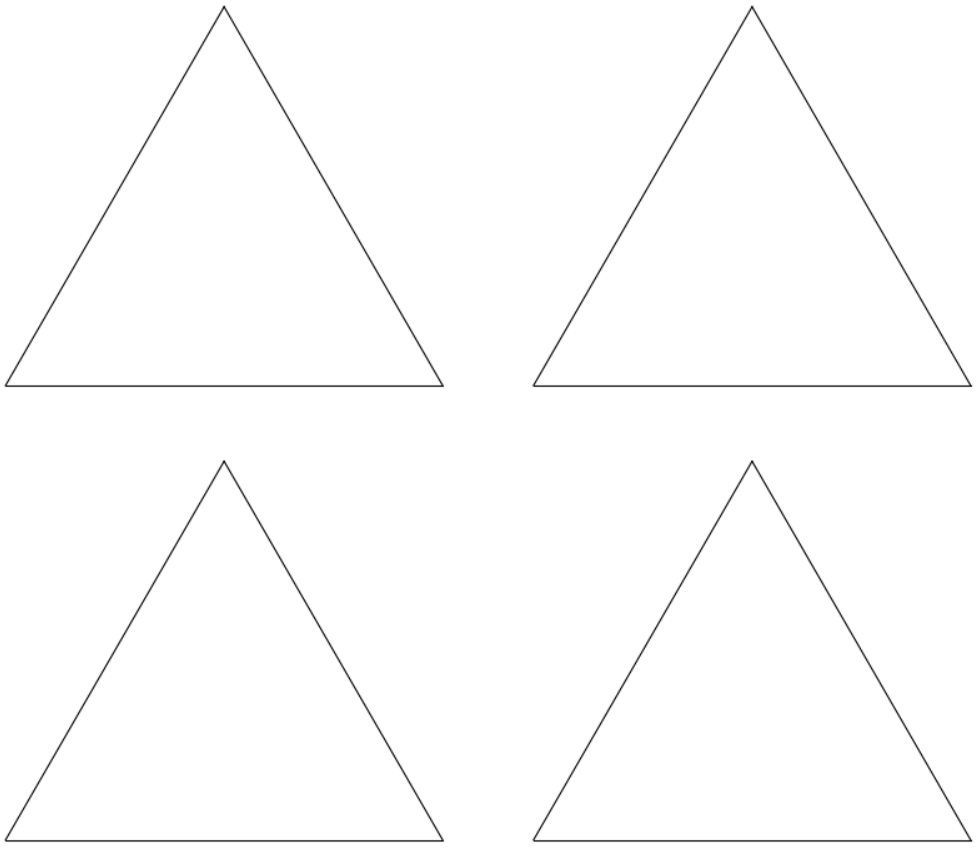
```
d = 1.0  # distance from centre
sig = 0.1  # scale for variance
# means for logspace (D=2)
means = np.array(np.meshgrid([-1, 1], [-1, 1])).T.reshape(-1, 2) * d
# means = np.array([(-d, -d), (d, -d), (-d, d), (d, d)])
covs = (  # covariance for logspace (D=2)
    np.array(
        [
            [[1, 0], [0, 1]],
            [[0.5, 0.15], [0.15, 0.5]],
            [[1.5, -1], [-1, 1.5]],
            [[1.2, -0.6], [-0.6, 1.2]],
        ]
    )
    * sig
)

means = ILRTransform().inverse_transform(means)  # compositional means (D=3)
size = 2000  # logo @ 10000
pts = [random_composition(mean=M, cov=C, size=size) for M, C in zip(means, covs)]

T = ILRTransform()
to_log = T.transform
from_log = T.inverse_transform

df = pd.DataFrame(np.vstack(pts))
df.columns = ["SiO2", "MgO", "FeO"]
df["Sample"] = np.repeat(np.arange(df.columns.size + 1), size).flatten()
chem = ["MgO", "SiO2", "FeO"]
```

```
fig, ax = plt.subplots(
    2, 2, figsize=(10, 10 * np.sqrt(3) / 2), subplot_kw=dict(projection="ternary")
)
ax = ax.flat
_ = [[x.set_ticks([]) for x in [a.taxis, a.laxis, a.raxis]] for a in ax]
```



First, let's look at the synthetic data itself in the ternary space:

```
kwargs = dict(alpha=0.2, s=3, no_ticks=True, axlabels=False)
for ix, sample in enumerate(df.Sample.unique()):
    comp = df.query(f"Sample == {sample}")
    comp.loc[:, chem].pyroplot.scatter(ax=ax[0], c=t10b3[ix], **kwargs)
plt.show()
```

We can take the mean and covariance in log-space to create covariance ellipses and vectors using principal component analysis:

```
kwargs = dict(ax=ax[1], transform=from_log, nstds=3)
ax[1].set_title("Covariance Ellipses and PCA Vectors")
for ix, sample in enumerate(df.Sample.unique()):
    comp = df.query(f"Sample == {sample}")
    tcomp = to_log(comp.loc[:, chem])
    plot_stdev_ellipses(tcomp.values, color=t10b3[ix], resolution=1000, **kwargs)
    plot_pca_vectors(tcomp.values, ls="-", lw=0.5, color="k", **kwargs)
plt.show()
```

We can also look at data density (here using kernel density estimation) in logratio-space:

```
kwargs = dict(ax=ax[-2], bins=100, axlabels=False)
ax[-2].set_title("Individual Density, with Contours")

for ix, sample in enumerate(df.Sample.unique()):
    comp = df.query(f"Sample == {sample}")
    comp.loc[:, chem].pyroplot.density(cmap="Blues", vmin=0.05, **kwargs)
    comp.loc[:, chem].pyroplot.density(
        contours=[0.68, 0.95],
        cmap="Blues_r",
        contour_labels={0.68: "", 0.95: "2"},
        **kwargs,
    )
plt.show()
```

```
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:241:␣
↪RuntimeWarning: invalid value encountered in log
  Y = np.log(X)  # Log operation
/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/comp/codata.py:241:␣
↪RuntimeWarning: invalid value encountered in log
  Y = np.log(X)  # Log operation
```

We can also do this for individual samples, and estimate percentile contours:

```
kwargs = dict(ax=ax[-1], axlabels=False)
ax[-1].set_title("Overall Density")
df.loc[:, chem].pyroplot.density(bins=100, cmap="Greys", **kwargs)
plt.show()
```

```
for a in ax:
    a.set_aspect("equal")
    a.patch.set_visible(False)
plt.show()
```

Total running time of the script: (0 minutes 2.258 seconds)

4.8.4 One Way to Do Ternary Heatmaps

There are multiple ways you can acheive ternary heatmaps, but those based on the cartesian axes (e.g. a regularly spaced rectangular grid, or even a regularly spaced triangular grid) can result in difficulties and data misrepresentation.

Here we illustrate how the ternary heatmaps for pyrolite are constructed using an irregualr triangulated grid and log transforms, and how this avoids some of the potential issues of the methods mentioned above.

Let's first get some data to deal with. `mpltern` has a conventient dataset which we can use here:

```
import numpy as np
import pandas as pd
from mpltern.datasets import get_scatter_points

np.random.seed(43)
df = pd.DataFrame(np.array([*get_scatter_points(n=80)]).T, columns=["A", "B", "C"])
df = df.loc[(df > 0.1).all(axis=1), :]
```

From this dataset we'll generate a *ternary_heatmap()*, which is the basis for ternary density diagrams via density():

```
from pyrolite.comp.codata import ILR, inverse_ILR
from pyrolite.plot.density.ternary import ternary_heatmap

coords, H, data = ternary_heatmap(
    df.values,
    bins=10,
    mode="density",
    remove_background=True,
    transform=ILR,
    inverse_transform=inverse_ILR,
    grid_border_frac=0.2,
)
```

This function returns more than just the coordinates and histogram/density estimate, which will come in handy for exploring how it came together. The data variable here is a dictonary with contains the grids and coordiantes used to construct the histogram/density diagram. We can use these to show how the ternary log-grid is constructed, and then transformed back to ternary space before being triangulated and interpoalted for the ternary heatmap:

```
import matplotlib.pyplot as plt

from pyrolite.util.math import flattengrid
from pyrolite.util.plot.axes import axes_to_ternary, share_axes

fig, ax = plt.subplots(3, 2, figsize=(6, 9))
ax = ax.flat

share_axes([ax[1], ax[2], ax[3]])
ax = axes_to_ternary([ax[0], ax[4], ax[5]])

ax[0].set_title("data", y=1.2)
df.pyroplot.scatter(ax=ax[0], c="k", alpha=0.1)
ax[0].scatter(*data["tern_bound_points"].T, c="k")

ax[1].set_title("transformed data", y=1.2)
ax[1].scatter(*data["tfm_tern_bound_points"].T, c="k")
ax[1].scatter(*data["grid_transform"](df.values).T, c="k", alpha=0.1)

ax[2].set_title("log grid", y=1.2)
ax[2].scatter(*flattengrid(data["tfm_centres"]).T, c="k", marker=".", s=5)
ax[2].scatter(*flattengrid(data["tfm_edges"]).T, c="k", marker=".", s=2)
ax[2].scatter(*data["tfm_tern_bound_points"].T, c="k")

ax[3].set_title("log-grid heatmap", y=1.2)
ax[3].pcolormesh(*data["tfm_edges"], H)
ax[3].scatter(*data["grid_transform"](df.values).T, c="white", alpha=0.8, s=1)

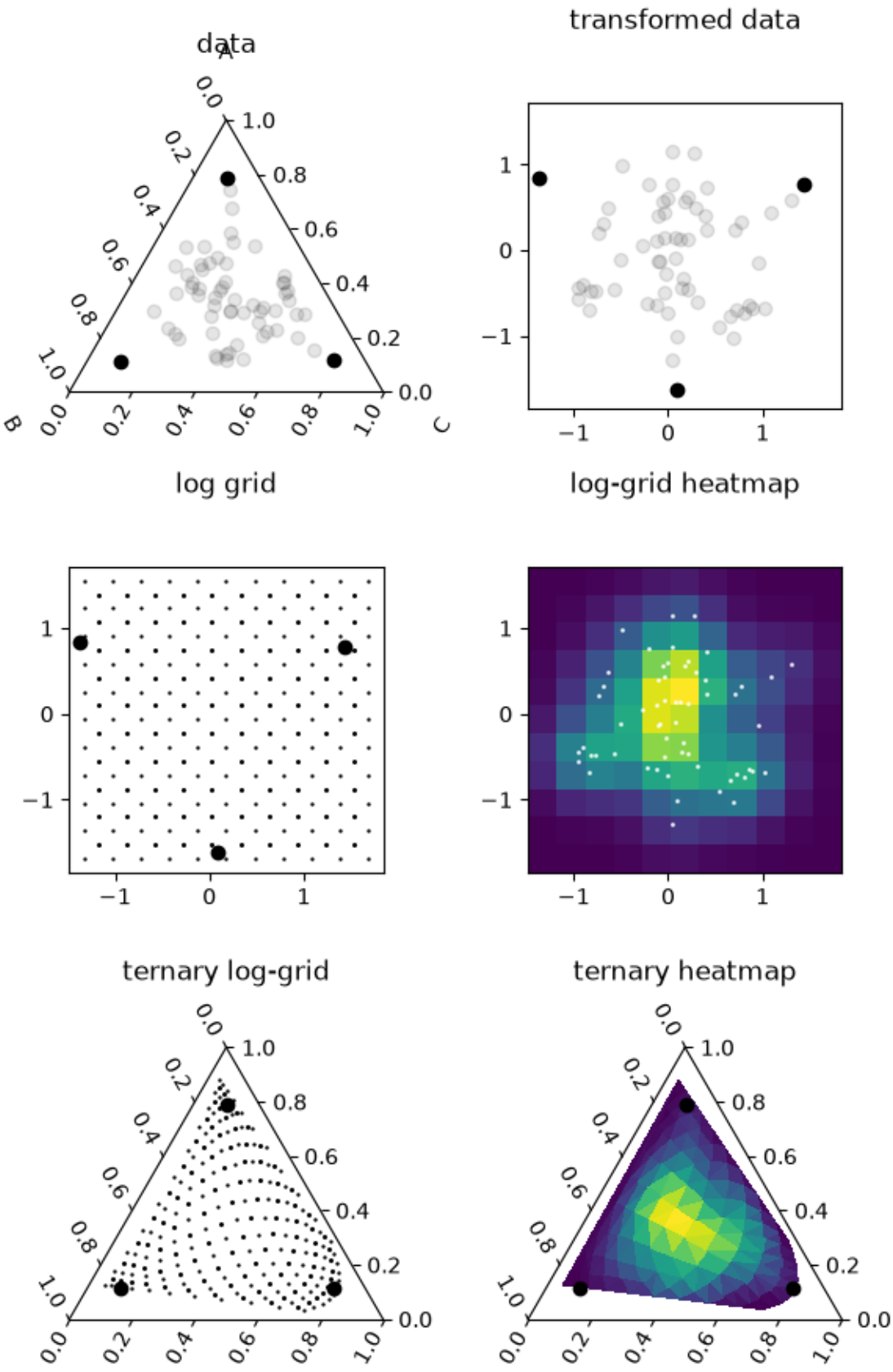
ax[4].set_title("ternary log-grid", y=1.2)
ax[4].scatter(*data["tern_centres"].T, c="k", marker=".", s=5)
ax[4].scatter(*data["tern_edges"].T, c="k", marker=".", s=2)
ax[4].scatter(*data["tern_bound_points"].T, c="k")
```

(continues on next page)

(continued from previous page)

```
ax[5].set_title("ternary heatmap", y=1.2)
ax[5].tripcolor(*coords.T, H.flatten())
ax[5].scatter(*data["tern_bound_points"].T, c="k")


plt.tight_layout()
```


data
A
B
C
transformed data
log grid
log-grid heatmap
ternary log-grid
ternary heatmap

```
plt.close("all")  # let's save some memory..
```

We can see how this works almost exactly the same for the histograms:

```
coords, H, data = ternary_heatmap(
    df.values,
    bins=10,
    mode="histogram",
    remove_background=True,
    transform=ILR,
    inverse_transform=inverse_ILR,
    grid_border_frac=0.2,
)
```

```
fig, ax = plt.subplots(3, 2, figsize=(6, 9))
ax = ax.flat

share_axes([ax[1], ax[2], ax[3]])
ax = axes_to_ternary([ax[0], ax[4], ax[5]])

ax[0].set_title("data", y=1.2)
df.pyroplot.scatter(ax=ax[0], c="k", alpha=0.1)
ax[0].scatter(*data["tern_bound_points"].T, c="k")

ax[1].set_title("transformed data", y=1.2)
ax[1].scatter(*data["tfm_tern_bound_points"].T, c="k")
ax[1].scatter(*data["grid_transform"](df.values).T, c="k", alpha=0.1)

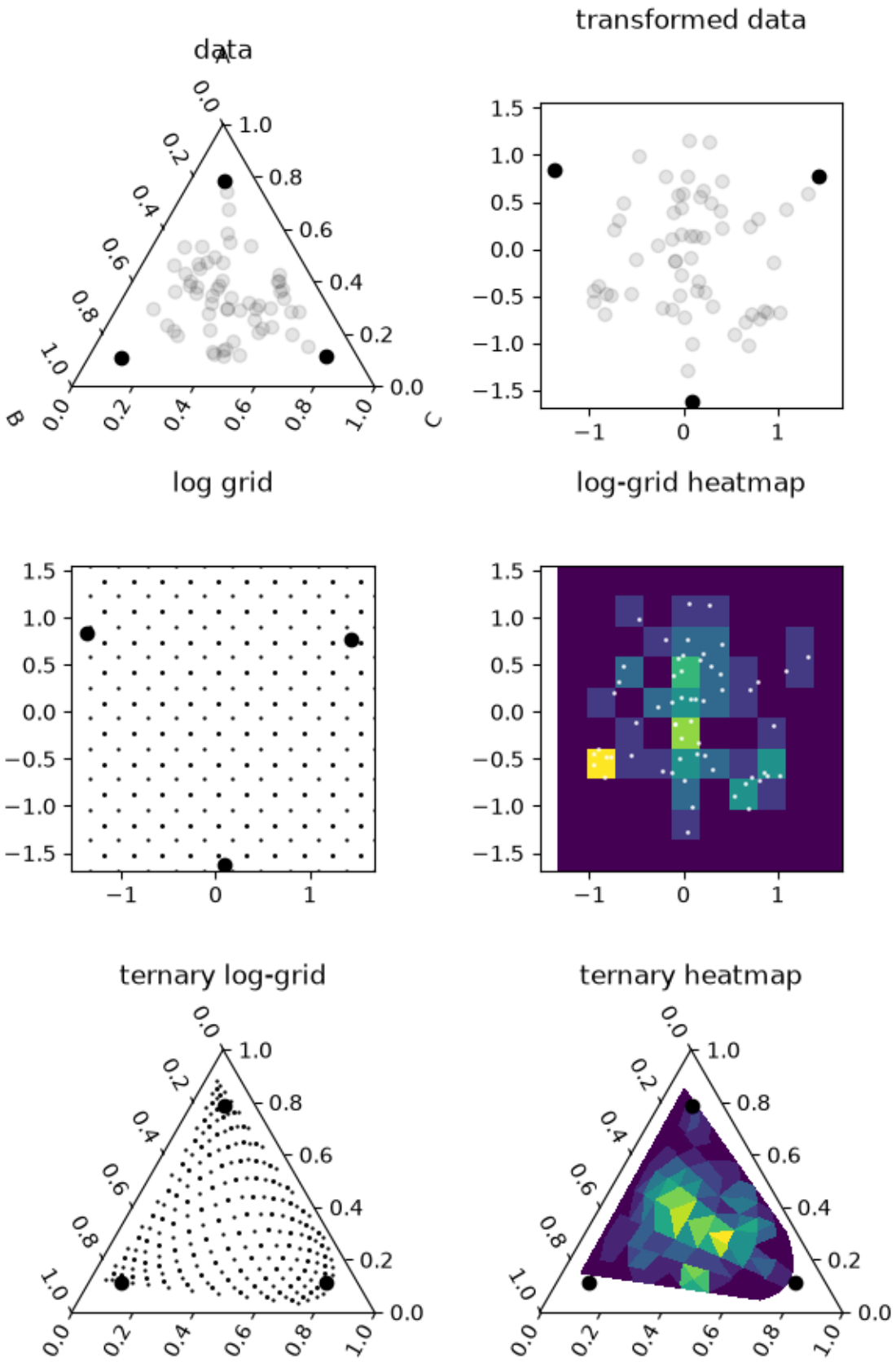
ax[2].set_title("log grid", y=1.2)
ax[2].scatter(*flattengrid(data["tfm_centres"]).T, c="k", marker=".", s=5)
ax[2].scatter(*flattengrid(data["tfm_edges"]).T, c="k", marker=".", s=2)
ax[2].scatter(*data["tfm_tern_bound_points"].T, c="k")

ax[3].set_title("log-grid heatmap", y=1.2)
ax[3].pcolormesh(*data["tfm_edges"], H)
ax[3].scatter(*data["grid_transform"](df.values).T, c="white", alpha=0.8, s=1)

ax[4].set_title("ternary log-grid", y=1.2)
ax[4].scatter(*data["tern_centres"].T, c="k", marker=".", s=5)
ax[4].scatter(*data["tern_edges"].T, c="k", marker=".", s=2)
ax[4].scatter(*data["tern_bound_points"].T, c="k")

ax[5].set_title("ternary heatmap", y=1.2)
ax[5].tripcolor(*coords.T, H.flatten())
ax[5].scatter(*data["tern_bound_points"].T, c="k")

plt.tight_layout()
```


data
A
B
C
transformed data
log grid
log-grid heatmap
ternary log-grid
ternary heatmap

Total running time of the script: (0 minutes 0.996 seconds)

4.9 Citation

{{ doibadages }}

If you use `pyrolite` extensively for your research, citation of the software would be particularly appreciated. It helps quantify the impact of the project (assisting those contributing through paid and volunteer work), and is one way to get the message out and help build the pyrolite community.

> **Note**
>
> `pyrolite` began as a personal project, but as the project develops there are likely to be other contributors. Check the *Contributors list* and add major contributors as authors. A CITATION.cff file has also been added to the repository, providing automated citation via GitHub.

`pyrolite` has been published in the Journal of Open Source Software (JOSS), and generally this paper is what you should cite. This paper was published in reference to `pyrolite` v0.2.7; if you're using a different version please note that in the citation.

While the exact format of your citation will vary with wherever you're publishing, it should take the general form:

> Williams et al., (2020). pyrolite: Python for geochemistry. Journal of Open Source Software, 5(50), 2314, doi: 10.21105/joss.02314

Or, if you wish to cite a specific version of the archive:

> Williams et al. ({{ year }}). pyrolite v {{ version }}, Zenodo, doi:10.5281/zenodo.2545106

If you're after a BibTeX citation for `pyrolite`, I've added one below.

```
@article{Williams2020,
  title = {pyrolite: Python for geochemistry},
  author = {Morgan J. Williams and
            Louise Schoneveld and
            Yajing Mao and
            Jens Klump and
            Justin Gosses and
            Hayden Dalton and
            Adam Bath and
            Steve Barnes},
  year = {2020},
  journal = {Journal of Open Source Software}
  doi = {10.21105/joss.02314},
  url = {https://doi.org/10.21105/joss.02314},
  publisher = {The Open Journal},
  volume = {5},
  number = {50},
  pages = {2314},
}
```

If you're using `pyrolite`'s implementation of `lambdas`, please consider citing a more recent publication directly related to this:

> Anenburg, M., & Williams, M. J. (2021). Quantifying the Tetrad Effect, Shape Components, and

Ce–Eu–Gd Anomalies in Rare Earth Element Patterns. Mathematical Geosciences. doi: 10.1007/s11004-021-09959-5

Or, if you're using BibTeX:

```
@article{Anenburg_Williams_2021,
  title={Quantifying the Tetrad Effect, Shape Components,
        and Ce-Eu-Gd Anomalies in Rare Earth Element Patterns},
  author={Anenburg, Michael and Williams, Morgan J.},
  year={2021},
  journal={Mathematical Geosciences},
  doi={10.1007/s11004-021-09959-5},
  url={https://doi.org/10.1007/s11004-021-09959-5},
  ISSN={1874-8953},
  month={Jul}
}
```

CHAPTER

FIVE

DEVELOPMENT

5.1 Changelog

All notable changes to this project will be documented here.

5.1.1 Development

> **Note**
>
> Changes noted in this subsection are to be released in the next version. If you're keen to check something out before its released, you can use a *development install* .

- **New Contributor / PR Merged**: Jarrod Burges contributed a PR to add magnetite trace element classification and discriminant diagrams; #129.

5.1.2 v0.3.9

- **Bugfix**: Colormapping options in *pyrolite.plot.color* should now work with pandas.arrays.StringArray.
- Updates for some deprecations in `scikit-learn`, matplotlib, pandas.
- Updated lock; released here more speedily than normal due to a wave of recent security updates.

5.1.3 v0.3.8

- **Bugfix**: Some fixes for using pandas.arrays.StringArray in e.g. *pyrolite.mineral.normative*.
- **Bugfix**: Some fixes for passing kwargs through density plot methods have been added; generally this has just subset the arguments passed to downstream functions to the obviously relevant ones.
- Updated some internal mechanics of *pyrolite.geochem.norm* and reference conditions, and associated transformation functions (e.g., where passing pandas.Series instead of pandas.DataFrame).
- Removed the unused function `pyrolite.util.log.stream_log()`.
- Removed `tinydb` dependency; files used in pyrolite are small and are instead simply stored as plaintext JSON in package data.
- Removed `legend.bbox_to_anchor : (1, 1)` from the pyrolite matplotlib style file; this was throwing warnings on import, inclusion of this to configure default legend location has been deprecated.
- Removed `numpydoc` dependency; some of the docstrings which had used this to dynamically present parameter options will now just point to where additional parameters are able to be used.

- Added dependabot config, added publishing GitHub action, updated binder config, updated GitHub action, and ReadTheDocs config.
- Added type hinting throughout.
- Migrated to the simpler MIT license.
- Updated package configuration; removed `MANIFEST.in` in favour of incorporation in `pyproject.toml`.

5.1.4 v0.3.7

- **New Contributor**: Jiawei Yun.
- **PR Merged**: Jiawei Yun contributed a PR to correct the FeO column label to Fe2O3T in the reference composition MUQ_Kabener2005; #109.
- **Docs**: Expanded the tutorial on creating plot templates in the documentation..
- **Docs**: Added citations and DOIs for reference composition descriptions in the documentation.
- **Docs**: Expanded the lattice strain example to include an example of fitting a profile, with a function added to the relevant module (*pyrolite.mineral.lattice.fit_lattice_strain()*).
- **Docs**: Expanded documentation around labelling schemes and selection of regions to plot for plot templates with an additional section on the relevant page.
- **Docs**: Removed QAP diagram from CIPW example (wt% vs vol% was a bit misleading, would need transformation to be accurate).a
- **Bugfix**: Updated *pyrolite.util.plot.interpolation.get_contour_paths()* to handle `matplotlib.contour.QuadContourSet`, and hence more recent `matplotlib` versions.
- **Bugfix**: Updated use of style file and built-in defaults to avoid hardcoding style (#107).
- **Bugfix**: Added handling for limit values below zero for spider plots with log scaled y axes. This should avoid unnecessary error messages.
- **Bugfix**: Reinstated imports of submodules so pandas accessors should be available immediately after `import pyrolite` (`df.pyrochem`, `df.pyrocomp`, `df.pyroplot`).
- **Feature**: Added new version of ICS International Chronostratigraphic Chart (2024-12; *pyrolite.util.time.Timescale*), and added the supereon level.
- Updated the feldspar ternary diagram orientation to fit convention as per DHZ (#108).
- Added a `fmt` kwarg to *pyrolite.util.skl.vis.plot_confusion_matrix()* to format values in the confusion matrix.
- Switched formatter to `ruff` from `black`.
- Moved to using `uv` as an environment management tool.

5.1.5 v0.3.6

- **PR Merged**: Sarah Shi contributed a PR to better handle values below zero during compositional renormalisation (to 1 or 100%; #104). The functions *pyrolite.comp.codata.close()* and *pyrolite.comp.codata.renormalise()* will now warn where values below zero exist, and replace these with `numpy.nan`.
- Added a `CITATION.cff` file to the repository.
- Various administrative changes (formatting, linting, meta-file management), minor bugfixes and addressing various deprecation/other warnings.

5.1.6 0.3.5

- **New Contributor**: Malte Mues
- **New Contributor**: Bob Myhill
- **Bugfix**: Fixed a bug with the index structure for the documentation example galleries.
- Expanded test suite for a text coverage bump.
- Started testing against and supporting Python 3.12.

pyrolite.geochem

- **PR Merged**: Malte Mues contributed a PR to allow series and dataframes with `Pm` data to be used with the `pyrolite.pyrochem` REE accessors (i.e., the column will not be dropped if it exists; #100). This PR also included an update for the `pyrolite.util.lambdas` functions to be more flexible in terms of access, including as individual series (allowing performant usage with pandarallel).
- Updates to `pyrolite.pyrochem` accessors to be more flexible, generally allowing usage with both DataFrame and Series objects (e.g. `df.pyrochem.REE`, `ser.pyrochem.REE`).

pyrolite.plot

- **PR Merged**: Bob Myhill contributed a series of pull requests to fix formatting of TAS diagram labels (#91), update the TAS field names in the JSON template (#92), improve the formatting and scaling of TAS diagrams (#93), add an option to add field labels in the visual centre of polygons (rather than the 'centroid'; #94), update some usage of `matplotlib` (#96), and allow selective plotting of individual fields when adding a classification diagram to axes (#98).

pyrolite.mineral

- Suppressed `pandas` performance warnings related to sequential construction of `pandas.DataFrames` in :func:`pyrolite.mineral.normative.CIPW_norm`.

5.1.7 0.3.4

- **Bugfix**: Tom Buckle contributed a PR with some minor bugfixes for the CIPW Norm (#87).
- Various maintenance updates, including migrating the package to use `pyproject.toml`.

5.1.8 0.3.3

- **New Contributor**: Sarah Shi
- **New Contributor**: Ondrej Lexa
- **Bugfix**: Updated docs builds to be compatible with recent versions of `sphinx-gallery`.
- **Bugfix**: Updated some `pandas` assignment, aggregation and similar operations to be compatible with more recent versions of Pandas.
- Added explicit Python 3.10 and 3.11 support.
- Removed Python 3.7 support (now end of life).

pyrolite.mineral

- **PR Merged**: Added an option to get expanded outputs for the CIPW Norm (from Tom Buckle; #80).
- **Bugfix**: Fixes and updated tests for CIPW Norm outputs.

pyrolite.plot

- **PR Merged**: Sarah Shi contributed a PR to add variations on the TAS diagram from Le Maitre (#79). These can be accessed by providing a `which_model` keyword argument to the *TAS* constructor (or plot template).
- **PR Merged**: Ondrej Lexa contributed a PR to add sandstone bulk geochemistry discrimination diagrams (*Pettijohn*, *Herron*; #82).
- **Bugfix**: Fixed issue with handling `vmin` and `vmax` for colormapping in *pyrolite.plot.color*.
- Suppressed warnings for 'division by zero'/'invalid value encountered in divide' in ternary diagram scatter plots.
- Added explicit support for colormapping categorical data in *pyrolite.plot.color*, such that ordering is preserved/consistent in e.g. legends.

pyrolite.util

- **Feature**: Added new version of ICS International Chronostratigraphic Chart (2022-10; *pyrolite.util.time.Timescale*).
- **Bugfix**: Corrected TAS diagram references, and fixed an issue where only the ID names were able to we added to diagrams.
- Updated axes-sharing utility function *share_axes()* to reflect more recent versions of `matplotlib`.
- Fixed issue in path interpolation for contours (*pyrolite.util.plot.interpolation.get_contour_paths*) for recent `matplotlib` versions.
- Updated figure export utility function to use `pathlib` syntax for suffixes, which should avoid potential for double suffixes (e.g. `figure_name.png.png`).

5.1.9 0.3.2

- **New Contributor**: Angela Rodrigues
- **Bugfix**: Edited docstrings and added ignore-warning for `numpydoc` warnings.
- **Bugfix**: Updated installation instructions and Binder configuration to use secure protocols when installing via `git` (i.e. https://)
- **Bugfix**: Update CI builds so that tests can be run on MacOS.

pyrolite.mineral

- **Feature**: Added a TAS-based iron correction following Middlemost (1989).
- **Bugfix**: Fixed some errors in mineral formulae and handling leading to inaccurate outputs from CIPW Norm.
- Split out volcanic from intrusive samples in the CIPW Norm volcanic rock comparison.
- Added SINCLAS abbreviations to the mineral dictionary associated with the CIPW Norm, so alternate mineral naming systems can be compared.

pyrolite.util

- **PR Merged**: Louise Schoneveld submitted a pull request to add bivariate and ternary classifier models for spinel compositions (*SpinelFeBivariate*, *SpinelTrivalentTernary*).
- **PR Merged**: Angela Rodrigues submitted a pull request to add the Jensen ternary cation classifier model for subalkalic volcanic rocks (*JensenPlot*).
- Updated *pyrolite.util.skl.vis.plot_confusion_matrix()* to be able to plot on existing axes, use an explicit class order and use rotation for e.g. long x-axis class label names.

- Updated references to `scipy.stats.gaussian_kde()` after namespace deprecation.

5.1.10 0.3.1

- **New Contributor**: Martin Bentley
- **New Contributor**: Chetan Nathwani
- **New Contributor**: Tom Buckle
- **New Contributor**: Nicolas Piette-Lauziere
- Removed a redundant `pathlib` dependency (which is standard library as of Python 3.4). This will fix an issue blocking setting up a `conda-forge` recipe (#51).
- Updated instances of redundant `numpy` types throughout to silence deprecation warnings (using base types `float`, `int` except where specific `numpy` types are required).
- Added a minimum `sympy` version requirement (v1.7) to avoid potential import errors.
- Updated minimum versions for `matplotlib` and `mpltern` to address potential version conflicts.
- A user installation is now recommended by default. This solves some potential issues on *-nix and MacOS systems.
- Fixed broken links to documentation in the README (thanks to Alessandro Gentilini).
- Fixed a bad documentation link the PyPI package information.
- Updated supported Python versions (Python 3.7-3.9).
- **Bugfix**: Updated use of `tinydb` databases to default to read-only access except where write access is explicitly needed. This should solve issues with permissions during installation and use of pyrolite on some systems (#61). Thanks to Antoine Ouellet for bringing this to attention, and both Sam Bradley and Alex Hunt for following up with the idea for the current solution.

pyrolite.geochem

- **Feature**: Nicolas Piette-Lauziere contributed two new functions for *pyrolite.geochem.alteration*: The chlorite-carbonate-pyrite index of Large et al. (2001; *CCPI()*) and the Alteration index of Ishikawa (1976; *IshikawaAltIndex()*).
- **Bugfix**: Fixed a bug where incomplete results where being returned for calls to `lambda_lnREE()` using the O'Neill algorithm. In this instance only the rows with the least missing data (typically those with no missing data) would return lambda values, other rows would contain null values. Thanks to Mark Pearce for identifying this one! In the meantime, using `df.pyrochem.lambda_lnREE(algorithm='opt')` will allow you to avoid the issue.
- **Bugfix**: Modified a few of the *pyrolite.geochem.pyrochem* methods to a avoid a bug due to assignment of the dataframe (*to_molecular()*, *to_weight()*, *recalculate_Fe()*). This bug seems to be contained to the dataframe accessor, the individual functions from *pyrolite.geochem.transform* appear to work as expected outside of this context. Thanks to Chetan Nathwani for highlighting this one!
- Renamed (private) package variables `__common_oxides__` and `__common_elements__` to `_common_oxides` and `_common_elements`

pyrolite.mineral

- **Feature**: CIPW function added to *pyrolite.mineral.normative*, largely from contributions by both Chetan Nathwani and Tom Buckle (#53). Note that the implementation still has a bug or two to be ironed out; it will currently raise a warning when used to make sure you're aware of this. An example has been added demonstrating

the intended functionality and demonstrating how coherent this is with existing implementations of CIPW (e.g. SINCLAS).

pyrolite.comp

- Updated *pyrolite.comp.codata.close()* to better deal with zeros (avoiding unnecessary warnings).
- Added spherical coordinate transformation to *pyrolite.comp.pyrocomp* and *pyrolite.comp.codata* (see *pyrolite.comp.pyrocomp.sphere()*).

pyrolite.plot

- **Feature**: Added ternary classification plot templates *USDASoilTexture*, `FeldsparTernary` and *QAP* (#49; idea and implementation of the latter thanks to Martin Bentley !). The idea for implementing the ternary diagram came from a discussion with Jordan Lubbers and Penny Wieser (of the Thermobar team, who are working in similar spaces); they've now implemented a version using `python-ternary` (rather than `mpltern`, which pyrolite is currently using).
- Updated examples and documentation for density and contour plots.
- Added autoscaling for standard *spider()* and related plots to address (#55)
- *process_color()* has been updated to better deal with data explicitly declared to be of a 'category' data type (as a `pandas.Series`), and also to better handle variation in mapping notations. Ordering of categorical variables will now be preserved during color-mapping.
- Added the option to have a 'bad' color to be used in categorical color-mapping where a category cannot be found.
- Inconsistent color specifications (e.g. a list or array of multiple types) will now result in an error when passed to *process_color()*.
- *parallel()* has been updated to align with other plotting functions (taking an optional `components` keyword argument).

pyrolite.util

- **Feature**: Added ternary classification models for *USDASoilTexture*, *FeldsparTernary* and *QAP* (#49; idea and implementation of the latter thanks to Martin Bentley).
- Added some functionality to *pyrolite.util.classification* to allow classifier fields to be precisely specified by ratios (useful in ternary systems), for multiple 'modes' of diagrams to be contained a single configuration file, and fixed some issues with labelling (arguments `add_labels` and `which_labels` can now be used to selectively add either field IDs/abbreviations or field names to classification diagrams).
- Limits are no longer explicitly required for bivariate templates (`xlim`, `ylim`) in *pyrolite.util.classification*.
- Update default parameterisation to `"full"` for lambdas, using all REE to generate orthogonal polynomial functions.
- Expanded *pyrolite.util.text.int_to_alpha()* to handle integers which are greater than 25 by adding multiple alphabetical characters (e.g. `26 > aa`), and to use the built-in `string.ascii_lowercase`.
- *save_figure()* will now create the directory it's given if it doesn't exist.
- Citation information for *lambdas* updated to include recent publications.
- Updated *plot_pca_vectors()* to accept line `colors` and `linestyles` arguments.
- Updated *init_spherical_octant()* to accept a `fontsize` argument.
- Added example for coloring ternary diagrams and ternary scatter points based on a ternary color system.

- Added helper for generating PCA component labels from a `scikit-learn` PCA object (`get_PCA_component_labels()`)
- Updated confusion matrix visualisation helper *`plot_confusion_matrix()`* to remove grid and provide more useful default colormap normalization options.
- Moved the manifold visualisation example to utility examples from plotting examples.
- Added a `fmt_string` argument to *`LogTransform`* for use in automated naming of transformed columns; this may be expanded to other transformers soon.
- Fixed some string issues for *`pyrolite.util.text`*.

5.1.11 0.3.0

- **New Contributor**: Lucy Mathieson
- Continuous Integration has been migrated from Travis to GitHub Actions.
- Added an `environment.yml` file for development environment consistency.
- Removed some tests dependent on `xlrd` due to external issues with reading `.xls` and `.xlsx` files with some OS-Python version combinations.
- Fixed some broken documentation links.
- Added `psutil` to requirements.

`pyrolite.plot`

- **Bugfix**: Fixed a bug where there scatter and line arguments would conflict for *`spider()`* (#46). To address this, *`spider()`* and related functions will now accept the keyword arguments `line_kw` and `scatter_kw` to explicitly configure the scatter and line aspects of the spider plot - enabling finer customization. An extra example has been added to the docs to illustrate the use of these parameters. Thanks go to Lucy Mathieson for raising this one!
- Added the `set_ticks` keyword argument to *`spider()`* and associated functions, allowing ticks to be optionally set (`set_ticks=False` if you don't want to set the x-ticks).
- Updated *`pyrolite.plot.color.process_color()`* to better handle colour mapping and added examples illustrating this. You can also now use RGBA colours when using the `color_mappings` keyword argument.
- Updated automated pyrolite `matplotlib` style export to be more reliable.
- Changed the default shading for *`density()`* to suppress error about upcoming `matplotlib` deprecation.
- Ordering for contours, contour names and contour styles is now preserved for *`density()`* and related functions.
- Updated `pyrolite.plot.templates.pearce` to use ratios from Sun & McDonough (1989), as in the Pearce (2008) paper.

`pyrolite.geochem`

- **Bugfix**: Fixed a bug where Eu was unnecessarily excluded from the *`lambda_lnREE()`* fit in all cases.
- **Bugfix**: Fixed a bug where ratio-based normalisation was not implemented for *`get_ratio()`* and related functions (#34)
- Added a local variable to *`pyrolite.geochem.ind`* to allow referencing of indexing functions (e.g. *`by_incompatibility()`*) by name, allowing easier integration with *`spider()`*.
- Added *`by_number()`* for indexing a set of elements by atomic number.

pyrolite.comp

- Updated the docstring for *pyrolite.comp.impute.EMCOMP()*.
- Minor updates for *pyrolite.comp.codata* labelling, and reflected changes in *pyrolite.util.skl.transform*. Issues were identified where the column name 'S' appears, and a workaround has been put in place for now.

pyrolite.util

- Expanded *pyrolite.util.lambdas* to allow fitting of tetrad functions, anomalies and estimation of parameter uncertainties for all three algorithms.
- Added *pyrolite.util.resampling* for weighted spatiotemporal bootstrap resampling and estimation, together with added a number of updates to *pyrolite.util.spatial* to provide required spatial-similarity functionality.
- Updated the geological timescale in *pyrolite.util.time* to use the 2020/03 version of the International Chronostratigraphic Chart (#45).
- Added *alphalabel_subplots()* for automatic alphabetic labelling of subplots (e.g. for a manuscript figure).
- Fixed an low-precision integer rollover issue in a combinatorial calculation for *pyrolite.util.missing* by increasing precision to 64-bit integers.
- Added *example_patterns_from_parameters()* to work with *pyrolite.util.lambdas* and generate synthetic REE patterns based on lambda and/or tetrad-parameterised curves.
- Moved `get_centroid()` from *pyrolite.util.classification* to *pyrolite.util.plot.helpers*
- Made a small change to *density* to allow passing contour labels as a list.
- *mappable_from_values()* will not accept a `norm` keyword argument, allowing use of colormap normalisers like `matplotlib.colors.Normalize`. This function was also updated to better handle `Series` objects.
- Fixed a small bug for *TAS* instantiation which didn't allow passing the variables to be used from a `pandas.DataFrame`. If you have different variable names, you can now pass them as a list with the `axes` keyword argument (e.g. `TAS(axes=['sio2', 'alkali'])`).
- Homogenised logging throughout the package - now all managed through *pyrolite.util.log*. The debugging and logging streaming function `stream_log()` can now also be accessed here (`pyrolite.util.log.stream_log()`).

5.1.12 0.2.8

- Updated citation information.
- Added specific testing for OSX for Travis, and updated the install method to better pick up issues with pip installations.
- **Feature**: Added a *gallery* of pages for each of the datasets included with `pyrolite`. This will soon be expanded, especially for the reference compositions (to address #38).

pyrolite.geochem

- **PR Merged**: Kaarel Mand submitted a pull request to add a number of shale and crustal compositions to the reference database.
- **Bugfix**: Fixed a bug where lambdas would only be calculated for rows without missing data. Where missing data was present, this would result in an assertion error and hence no returned values.

- **Bugfix**: Fixed a bug where missing data wasn't handled correctly for calculating lambdas. The functions now correctly ignore the potential contribution of elements which are missing when parameterising REE patterns. Thanks to Steve Barnes for the tip off which led to identifying this issue!
- **Feature**: Added *pyrolite.geochem.ind.REY()*, *list_REY()*, and *REY()* to address (#35). This issue was also opened by Kaarel Mand!
- As a lead-in to a potential change in default parameterisation, you can now provide additional specifications for the calculation of `lambdas` to *lambda_lnREE()* and *calc_lambdas()* to determine the basis over which the individual orthogonal polynomials are defined (i.e. which REE are included to define the orthonormality of these functions). For the keyword argument `params`, (as before) you can pass a list of tuples defining the constants representing the polynomials, but you can now alternatively pass the string `"ONeill2016"` to explicitly specify the original parameterisation, or `"full"` to use all REE (including Eu) to define the orthonormality of the component functions (i.e. using `params="full"`). To determine which elements are used to perform the *fit*, you can either filter the columns passed to these functions or specifically exclude columns using the `exclude` keyword argument (e.g. the default remains `exclude=["Eu"]` which excludes Eu from the fitting process). Note that the default for fitting will remain, but going forward the default for the definition of the polynomial functions will change to use all the REE by default (i.e. change to `params="full"`).
- Significant performance upgrades for *lambda_lnREE()* and associated functions (up to 3000x for larger datasets).
- Added a minimum number of elements, configurable for *lambda_lnREE()*. This is currently set to seven elements (about half of the REE), and probably lower than it should be ideally. If for some reason you want to test what lambdas (maybe just one or two) look like with less elements, you can use the `min_elements` keyword argument.
- Added *list_isotope_ratios()* and corresponding selector *isotope_ratios()* to subset isotope ratios.
- Added *parse_chem()* to translate geochemical columns to a standardised (and pyrolite-recognised) column name format.

pyrolite.plot

- **Bugfix**: Fixed a bug where arguments processing by *pyrolite.plot.color* would consume the 'alpha' parameter if no colour was specified (and as such it would have no effect on the default colors used by `pyplot`)
- **Bugfix**: *pyrolite.plot.color* now better handles colour and value arrays.
- **Bugfix**: Keyword arguments passed to *pyrolite.plot.density* will now correctly be forwarded to respective functions for histogram and hexbin methods.
- Customised `matplotlib` styling has been added for `pyrolite` plotting functions, including legends. This is currently relatively minimal, but could be expanded slightly in the future.
- The `bw_method` argument for `scipy.stats.kde.gaussian_kde()` can now be parsed by `pyrolite` density-plot functions (e.g. *density()*, *heatscatter()*). This means you can modify the default bandwidth of the gaussian kernel density plots. Future updates may allow non-Gaussian kernels to also be used for these purposes - keep an eye out!
- You can now specify the y-extent for conditional spider plots to restrict the range over which the plot is generated (and focus the plot to where your data actually is). For this, feed in a `(min, max)` tuple for the `yextent` keyword argument.
- The `ybins` argument for *spider()* and related functions has been updated to `bins` to be in line with other functions.
- Conditional density *REE()* plots now work as expected, after some fixes for generating reverse-ordered indexes and bins
- Added a filter for ternary density plots to ignore true zeroes.

- Some updates for *pyrolite.plot.color* for alpha handling and colour arrays .

pyrolite.comp

- Updated transform naming to be consistent between functions and class methods. From this version use capitalised versions for the transform name acronyms (e.g. `ILR` instead of `ilr`).
- Added for transform metadata storage within DataFrames for *pyrocomp*, and functions to access transforms by name.
- Added labelling functions for use with *pyrolite.comp.pyrocomp* and *codata* to illustrate the precise relationships depicted by the logratio metrics (specified using the `label_mode` parameter supplied to each of the resepective *pyrocomp* logratio transforms).

pyrolite.util

- Revamped *pyrolite.util.classification* to remove cross-compatibility bugs with OSX/other systems. This is now much simpler and uses JSON for serialization.
- Small fix for *mappable_from_values()* to deal with NaN values.
- Added *pyrolite.util.log* for more streamlined logging (from `pyrolite-meltsutil`)
- Added *pyrolite.util.spatial.levenshtein_distance()* for comparing sequence differences/distances between 1D iterables (e.g. strings, lists).

5.1.13 0.2.7

- Bugfix to include radii data in MANIFEST.in

5.1.14 0.2.6

- **New Contributors**: Kaarel Mand and Laura Miller
- **PR Merged**: Louise Schoneveld submitted a pull request to fill out the newly-added Formatting and Cleaning Up Plots tutorial. This tutorial aims to provide some basic guidance for common figure and axis formatting tasks as relevant to `pyrolite`.
- Added `codacy` for code quality checking, and implemented numerous clean-ups and a few new tests across the package.
- Performance upgrades, largely for the documentation page. The docs page should build and load faster, and have less memory hang-ups - due to smaller default image sizes/DPI.
- Removed dependency on `fancyimpute`, instead using functions from `scikit-learn`

pyrolite.geochem

- **Bugfix**: pyrolite lambdas differ slightly from[1] (#39). Differences between the lambda coefficients of the original and pyrolite implementations of the lambdas calculation were identified (thanks to Laura Miller for this one). With further investigation, it's likely the cost function passed to `scipy.optimize.least_squares()` contained an error. This has been remedied, and the relevant pyrolite functions now by default should give values comparable to[1]. As part of this, the reference composition `ChondriteREE_ON` was added to the reference database with the REE abundances presented in[1].
- **Bugfix**: Upgrades for *convert_chemistry()* to improve performance (#29). This bug appears to have resulted from caching the function calls to *pyrolite.geochem.ind.simple_oxides()*, which is addressed with 18fede0.

[1] O'Neill, H.S.C., 2016. The Smoothness and Shapes of Chondrite-normalized Rare Earth Element Patterns in Basalts. J Petrology 57, 1463–1508. doi: 10.1093/petrology/egw047.

- **Feature**: Added the[2] ionic radii for use in silicate geochemistry (#41), which can optionally be used with *pyrolite.geochem.ind.get_ionic_radii()* using the `source` keyword argument (`source='Whittaker'`). Thanks to Charles Le Losq for the suggestion!
- **Bugfix**: Removed an erroneous zero from the GLOSS reference composition (`GLOSS_P2014` value for Pr).
- Updated *REE()* to default to `dropPm=True`
- Moved `pyrolite.mineral.ions` to *pyrolite.geochem.ions*

pyrolite.mineral

- **Bugfix**: Added the mineral database to `MANIFEST.in` to allow this to be installed with `pyrolite` (fixing a bug where this isn't present, identified by Kaarel Mand).

pyrolite.plot

- **Bugfix**: Updated *pyrolite.plot* to use `pandas.DataFrame.reindex()` over `pandas.DataFrame.loc()` where indexes could include missing values to deal with #31.
- Updated *spider()* to accept `logy` keyword argument, defaulting to `True`

pyrolite.util

- Broke down *pyrolite.util.plot* into submodules, and updated relevant imports. This will result in minimal changes to API usage where functions are imported explicitly.
- Split out *pyrolite.util.lambdas* from *pyrolite.util.math*
- Added a minimum figure dimension to *init_axes()* to avoid having null-dimensions during automatic figure generation from empty datasets.
- Added *example_spider_data()* to generate an example dataset for demonstrating spider diagrams and associated functions. This allowed detailed synthetic data generation for *spider()* and *pyrolite.plot.pyroplot.REE()* plotting examples to be cut down significantly.
- Removed unused submodule `pyrolite.util.wfs`

5.1.15 0.2.5

- **PR Merged**: @lavender22 updated the spider diagram example to add a link to the normalisation example (which lists different reservoirs you can normalise to).
- Added an 'Importing Data' section to the docs *Getting Started page*.
- Disabled automatic extension loading (e.g. for `pyrolite_meltsutil`) to avoid bugs during version mismatches.

pyrolite.comp

- Updated the *pyrolite.comp.pyrocomp* dataframe accessor API to include reference to compositional data log transform functions within *pyrolite.comp.codata*

pyrolite.plot

- Added support for spider plot index ordering added with the keyword `index_order` (#30)
- Added support for color indexing in *color* using `pandas.Series`, and also for list-like arrays of categories

[2] Whittaker, E.J.W., Muntus, R., 1970. Ionic radii for use in geochemistry. Geochimica et Cosmochimica Acta 34, 945–956. doi: 10.1016/0016-7037(70)90077-3.

- Added a workaround for referring to axes positions where the projection is changed to a ternary projection (displacing the original axis), but the reference to the original axes object (now booted from `fig.axes`/`fig.orderedaxes`) is subsequently used.
- Updated *process_color()* processing of auxillary color keyword arguments (fixing a bug for color arguments in *stem()*)
- Added support for a `color_mappings` keyword argument for mapping categorical variables to specific colors.
- Updated the effect of `relim` keyword argument of *density()* to remove the scaling (it will no longer log-scale the axes, just the grid/histogram bins).
- Updated `Grid` to accept an x-y tuple to specify numbers of bins in each direction within a grid (e.g. `bins=(20, 40)`)
- Updated the grids used in some of the *density()* methods to be edges, lining up the arrays such that shading parameters will work as expected (e.g. `shading='gouraud'`)

pyrolite.geochem

- Added sorting function `~pyrolite.geochem.ind.by_incompatibility` for incompatible element sorting (based on BCC/PM relative abundances).

pyrolite.mineral

- Minor bugfix for *update_database()*

pyrolite.util

- Moved `check_perl()` out of `pyrolite` into `pyrolite_meltsutil`

5.1.16 0.2.4

- Removed Python 3.5 support, added Python 3.8 support.
- Updated ternary plots to use `mpltern` (#28)
- Added a ternary heatmap tutorial

pyrolite.plot

- Added *pyrolite.plot.pyroplot.plot()* method
- Removed `pyrolite.plot.pyroplot.ternary()` method (ternary plots now served through the same interface as bivariate plots using *pyrolite.plot.pyroplot.scatter()*, *pyrolite.plot.pyroplot.plot()*, and *pyrolite.plot.pyroplot.plot()*)
- Added *pyrolite.plot.color* for processing color arguments.
- Moved *pyrolite.plot.density* to its own sub-submodule, including *pyrolite.plot.density.ternary* and *pyrolite.plot.density.grid*

pyrolite.util

- Updated *time* to include official colors.
- Added *pyrolite.util.time* example
- Updated `stream_log()` to deal with logger duplication issues.
- Various updates to *pyrolite.util.plot*, noted below:

- Added universal axes initiation for bivariate/ternary diagrams using `init_axes()` and axes labelling with `label_axes()`,
- Added keyword argument processing functions `scatterkwargs()`, `linekwargs()`, and `patchkwargs()`
- Added functions for replacing non-projected axes with ternary axes, including `replace_with_ternary_axis()`, `axes_to_ternary()` (and `get_axes_index()` to maintain ordering of new axes)
- Added `get_axis_density_methods()` to access the relevant histogram/density methods for bivariate and ternary axes
- Renamed private attributes for default colormaps to `DEFAULT_DISC_COLORMAP` and `DEFAULT_CONT_COLORMAP`
- Updated `add_colorbar()` to better handle colorbars for ternary diagrams

5.1.17 0.2.3

- Added *Getting Started page*

pyrolite.mineral

- Updated database for *pyrolite.mineral.mindb* to include epidotes, garnets, micas

pyrolite.plot

- Minor updates for *pyrolite.plot.templates*, added functionality to *pyrolite.plot.templates.TAS()* stub.
- Fixed a bug for `vmin` in *pyrolite.plot.spider* density modes

pyrolite.geochem

- *pyrolite.geochem.parse* now also includes functions which were previously included in `pyrolite.geochem.validate`
- Fixed some typos in reference compositions from Gale et al. (2013)

pyrolite.util

- Added `pyrolite.util.plot.set_ternary_labels()` for setting and positioning ternary plot labels

5.1.18 0.2.2

pyrolite.geochem

- Added *SCSS()* for modelling sulfur content at sulfate/sulfide saturation.

pyrolite.mineral

- Added *mineral database* and and *mineral endmember decomposition* examples

5.1.19 0.2.1

- Updated and refactored documentation
  - Added *Development*, Debugging section, *Extensions*
  - Added `sphinx_gallery` with binder links for examples
  - Removed duplicated examples

  - Amended *citation guidelines*

- Removed extensions from pyrolite (`pyrolite.ext.datarepo`, `pyrolite.ext.alphamelts`). These will soon be available as separate extension packages. This enabled faster build and test times, and removed extraneous dependencies for the core `pyrolite` package.
- Added `stats_require` as optional requirements in `setup.py`

pyrolite.geochem

- Added *get_ratio()* and *pyrolite.geochem.pyrochem.get_ratio()*
- Added *pyrolite.geochem.pyrochem.compositional()* selector

pyrolite.plot

- *parallel()* now better handles `pyplot` figure and subplot arguments
- `ternary()` and related functions now handle label offsets and label fontsizes
- Minor bugfixes for *density*
- Added `unity_line` argument to *spider()* to be consistent with *REE_v_radii()*

pyrolite.mineral

- Added a simple *pyrolite.mineral.mindb* database
- Added *pyrolite.mineral.transform* to house mineral transformation functions
- Expanded *pyrolite.mineral.normative* to include *unmix()* and *pyrolite.mineral.normative.endmember_decompose()* for composition-based mineral endmember decomposition

pyrolite.util

- Added `pyrolite.util.plot.mappable_from_values()` to enable generating `ScalarMappable` objects from an array of values, for use in generating colorbars

5.1.20 0.2.0

- Added alt-text to documentation example images
- Updated contributing guidelines
- Added Python 3.8-dev to Travis config (not yet available)
- Removed `pandas-flavor` decorators from *pyrolite.geochem* and *pyrolite.comp*, eliminating the dependency on `pandas-flavor`

pyrolite.geochem

- Expanded *pyrolite.geochem.pyrochem* DataFrame accessor and constituent methods
- Updates and bugfixes for *pyrolite.geochem.transform* and *pyrolite.geochem.norm*
- Updated the *normalization example*

pyrolite.comp

- Added *pyrolite.comp.pyrocomp* DataFrame accessor with the *pyrolite.comp.codata.renormalise()* method.
- Removed unused imputation and aggregation functions.

pyrolite.plot

- Added *heatscatter()* and *example*.
- Updates and bugfixes for *pyrolite.plot.spider.REE_v_radii()*, including updating spacing to reflect relative ionic radii

pyrolite.util

- Added `pyrolite.util.plot.get_twins()`

5.1.21 0.1.21

pyrolite.plot

- Added parallel coordinate plots: *pyrolite.plot.pyroplot.parallel()*
- Updated *scatter()* and `ternary()` to better deal with colormaps

pyrolite.ext.alphamelts

- Updated `pyrolite.ext.alphamelts` interface:
  - Docs
  - Updated to default to tables with percentages (Wt%, Vol%)
  - Updated `plottemplates` y-labels
  - Fixed `automation` grid bug

5.1.22 0.1.20

pyrolite.geochem

- Stub for *pyrolite.geochem.pyrochem* accessor (yet to be fully developed)
- Convert reference compositions within of *pyrolite.geochem.norm* to use a JSON database

pyrolite.util.skl

- Added *pyrolite.util.skl.vis.plot_mapping()* for manifold dimensional reduction
- Added *pyrolite.util.skl.vis.alphas_from_multiclass_prob()* for visualising multi-class classification probabilities in scatter plots

pyrolite.plot

- Added *pyrolite.plot.biplot* to API docs
- Updated default y-aspect for ternary plots and axes patches

pyrolite.ext.alphamelts

- Updated `pyrolite.ext.alphamelts.automation`, `pyrolite.ext.alphamelts.meltsfile`, `pyrolite.ext.alphamelts.tables`
- Updated docs to use `pyrolite.ext.alphamelts.automation.MeltsBatch` with a parameter grid

5.1.23 0.1.19

- Added this changelog
- Require `pandas` >= v0.23 for DataFrame accessors

pyrolite.geochem

- Moved normalization into *pyrolite.geochem*
- Improved support for molecular-based calculations in *pyrolite.geochem*
- Added *pyrolite.geochem* section to API docs
- Added the `convert_chemistry()` docs example

pyrolite.ext.alphamelts

- Improvements for `pyrolite.ext.alphamelts.download`
- Completed `pyrolite.ext.alphamelts.automation.MeltsBatch`
- Added the `pyrolite.ext.alphamelts.web` docs example
- Added `pyrolite.ext.alphamelts.plottemplates` to API docs
- Added `pyrolite.ext.alphamelts.tables.write_summary_phaselist()`
- Added `pyrolite.ext.alphamelts.automation.exp_name()` for automated alphaMELTS experiment within batches

pyrolite.util

- Added `pyrolite.util.meta.ToLogger` output stream for logging
- Added `pyrolite.util.multip.combine_choices()` for generating parameter combination grids

5.1.24 0.1.18

- Require `scipy` >= 1.2

pyrolite.plot

- Automatic import of dataframe accessor `df.pyroplot` removed; import *pyrolite.plot* to use *pyrolite.plot.pyroplot* dataframe accessor
- Updated label locations for *pyrolite.plot.biplot*
- Default location of the y-axis updated for *pyrolite.plot.stem.stem()*

pyrolite.geochem

- Added stub for `pyroilte.geochem.qualilty`

pyrolite.util

- Moved `pyrolite.classification` to *pyrolite.util.classification*
- Added `pyrolite.util.plot.marker_cycle()`

5.1.25 0.1.17

- Update status to Beta

pyrolite.geochem

- Added database for geochemical components (`geochemdb.json`) for faster import via *common_elements()* and *common_oxides()*
- Added stub for *pyrolite.geochem.isotope*
- Update to using `pyrolite.util.transform.aggregate_element()` rather than `aggregate_cation`

pyrolite.plot

- Expanded use of *pyrolite.plot.pyroplot* dataframe accessor
- Added `pyrolite.plot.pyrochem.cooccurence()`
- Added *pyrolite.plot.biplot*
- Added support for conditional density spiderplots within *spider()* and *REE_v_radii()*
- Updated keyword argument parsing for *spider()*

pyrolite.mineral

- Removed automatic import of mineral structures to reduce delay
- Updated *pyrolite.mineral.lattice.strain_coefficient()*
- Added stub for *pyrolite.mineral.normative()*
- Updated *pyrolite.mineral.sites.Site*

pyrolite.util

- Added functions for interpolating paths and patches (e.g. contours) and exporting these: `interpolate_path()`, `interpolated_patch_path()`, `get_contour_paths()`, `path_to_csv()`
- Added `util.plot._mpl_sp_kw_split()`
- Added `util.text.remove_suffix()`
- Added `util.text.int_to_alpha()`

pyrolite.ext

- Updated alphaMELTS interface location to external package interface rather than utility (from *pyrolite.util* to `pyrolite.ext`)
- Added `pyrolite.ext.datarepo` stub

5.1.26 0.1.16

pyrolite.mineral

- Added *pyrolite.mineral.lattice* example
- Added *pyrolite.mineral.lattice.youngs_modulus_approximation()*

pyrolite.ext.alphamelts

- Added `pyrolite.ext.alphamelts` Monte Carlo uncertainty estimation example
- Added `pyrolite.ext.alphamelts.automation.MeltsExperiment.callstring()` to facilitate manual reproducibility of pyrolite calls to alphaMELTS.
- Improved alphaMELTS interface termination
- Added `pyrolite.ext.alphamelts.plottemplates.phase_linestyle()` to for auto-differentiated linestyles in plots generated from alphaMELTS output tables
- Added `pyrolite.ext.alphamelts.plottemplates.table_by_phase()` to generate axes per phase from a specific output table

pyrolite.geochem

- Added MORB compositions from Gale et al. (2013) to Reference Compositions
- Updated `pyrolite.geochem.ind.get_radii` to *pyrolite.geochem.ind.get_ionic_radii()*
- `dropPm` parameter added to *pyrolite.geochem.ind.REE()*

pyrolite.plot

- Updated `pyrolite.plot.spider.REE_radii_plot` to *pyrolite.plot.spider.REE_v_radii()*
- Updated `pyrolite.util.meta.steam_log()` to take into account active logging handlers

pyrolite.util

- Added *pyrolite.util.pd.drop_where_all_empty()*
- Added *pyrolite.util.pd.read_table()* for simple `.csv` and `.xlsx`/`.xls` imports
- Added `pyrolite.util.plot.rect_from_centre()`
- Added *pyrolite.util.text.slugify()* for removing spaces and non-alphanumeric characters

5.1.27 0.1.15

pyrolite.ext.alphamelts

- Bugfixes for `automation` and `download`
- Add a `permissions` keyword argument to *pyrolite.util.general.copy_file()*

5.1.28 0.1.14

- Added Contributor Covenant Code of Conduct

pyrolite.plot

- Added *pyrolite.plot.stem.stem()* example
- Added *pyrolite.plot.stem*
- Added *pyrolite.plot.stem* to API docs
- Added *pyrolite.plot.stem* example

pyrolite.mineral

- Added *pyrolite.mineral.lattice* for lattice strain calculations
- Added *pyrolite.mineral* to API docs

pyrolite.ext.alphamelts

- Improved `pyrolite.ext.alphamelts.automation` workflows, process tracking and termination
- Incorporated `MeltsProcess` into `MeltsExperiment`
- Added `MeltsBatch` stub
- Added `read_meltsfile()` and `read_envfile()`
- Added `pyrolite.ext.alphamelts.plottemplates`
- Added `pyrolite.ext.alphamelts.tables.get_experiments_summary()` for aggregating alphaMELTS experiment results across folders

pyrolite.util

- Added manifold uncertainty example with *pyrolite.util.skl.vis.plot_mapping()*
- Updated `pyrolite.util.ditributions.norm_to_lognorm`
- Added `pyrolite.util.general.get_process_tree()` to extract related processes
- Added *pyrolite.util.pd.zero_to_nan()*

5.1.29 0.1.13

pyrolite.ext.alphamelts

- Updated `pyrolite.ext.alphamelts.automation.MeltsProcess` workflow
- Updated `pyrolite.ext.alphamelts.download` local installation
- Added `pyrolite.ext.alphamelts.install` example
- Added `pyrolite.ext.alphamelts.tables` example
- Added `pyrolite.ext.alphamelts.automation` example
- Added `pyrolite.ext.alphamelts.env` example

5.1.30 0.1.12

pyrolite.util.pd

- Bugfix for *pyrolite.util.pd.to_frame()*

5.1.31 0.1.11

- Added *citation* page to docs
- Added *contributors* page to docs
- Updated docs *future* page
- Updated docs config and logo

pyrolite.geochem

- Added stub for *pyrolite.geochem.isotope*, pyrolite.geochem.isotope.count

pyrolite.comp

- Added compositional data example
- Added *pyrolite.comp.codata.logratiomean()*
- Added pyrolite.data.Aitchison and assocaited data files

pyroilite.ext.alphamelts

- Added pyrolite.ext.alphamelts to API docs
- Added pyrolite.ext.alphamelts.automation

pyrolite.util

- Expanded *pyrolite.util* API docs
- Added *pyrolite.util.distributions*
- Moved pyrolite_datafolder from *pyrolite.util.general* to *pyrolite.util.meta.pyrolite_datafolder()*
- Added share_axes(), ternary_patch(), subaxes()
- Added *pyrolite.util.units*, moved pyrolite.geochem.norm.scale_multiplier to *pyrolite.util.units.scale()*
- Updated *pyrolite.util.synthetic.random_cov_matrix()* to optionally take a sigmas keyword argument

5.1.32 0.1.10

- Updated *installation* docs

pyrolite.util

- Added *pyrolite.util.types*
- Added *pyrolite.util.web*
- Added manifold uncertainty example with *pyrolite.util.skl.vis.plot_mapping()*
- Moved stream log to pyrolite.util.meta.stream_log()
- Added *pyrolite.util.meta.take_me_to_the_docs()*
- Updated *pyrolite.util.skl.vis*

pyrolite.ext.datarepo

- Updated pyrolite.ext.datarepo.georoc (then pyrolite.util.repositories.georoc)

5.1.33 0.1.9

pyrolite.plot

- Added *pyrolite.plot.templates*, and related API docs
- Added Pearce templates under pyrolite.plot.templates.pearce

- Update default color schemes in scatter plots within *pyrolite.plot* to fall-back to matplotlib.pyplot cycling

pyrolite.util

- Added conditional import for PCA and statsmodels within *pyrolite.util.plot*
- Refactored sklearn utilities to submodule *pyrolite.util.skl*
- Added *pyrolite.util.meta.sphinx_doi_link()*
- Updated *pyrolite.util.meta.inargs()*
- Updated pyrolite.util.meta.stream_log() (then pyrolite.util.general.stream_log)
- Added conditional import for imblearn under *pyrolite.util.skl.pipeline*

pyrolite.ext.alphamelts

- Added pyrolite.ext.alphamelts (then pyrolite.util.alphamelts)
- Bugfix for Python 3.5 style strings in pyrolite.ext.alphamelts.parse

5.1.34 0.1.8

- Bugfixes for *pyrolite.plot.spider* and pyrolite.util.plot.conditional_prob_density

5.1.35 0.1.7

pyrolite.plot

- Added *cooccurence()* method to *pyrolite.plot.pyroplot* DataFrame accessor

pyrolite.util

- Added *pyrolite.util.missing.cooccurence_pattern()*
- Moved pyrolite.util.skl.plot_cooccurence to pyrolite.util.plot.plot_cooccurence()
- Updated pyrolite.util.plot.conditional_prob_density(), pyrolite.util.plot.bin_edges_to_centres() and pyrolite.util.plot.bin_centres_to_edges()

5.1.36 0.1.6

pyrolite.plot

- Update *spider()* to use contours keyword argument, and pass these to pyrolite.util.plot.plot_Z_percentiles()

pyrolite.util

- Bugfixes for invalid steps in *pyrolite.util.math.linspc_()*, *pyrolite.util.math.logspc_()*

5.1.37 0.1.5

- Updated docs *future* page

pyrolite.geochem

- Bugfix for iron redox recalcuation in *pyrolite.geochem.transform.convert_chemistry()*

pyrolite.plot

- Added `mode` keyword argument to *pyrolite.plot.spider.spider()* to enable density-based visualisation of spider plots.
- Update *pyrolite.plot.pyroplot.spider()* to accept `mode` keyword argument
- Update *pyrolite.plot.pyroplot.REE()* to use a `index` keyword arguument in the place of the previous `mode`; `mode` is now used to switch between line and density base methods of visualising spider plots consistent with *spider()*
- Added *spider()* *examples for conditional density plots* using `conditional_prob_density()`
- Bugfix for `set_under` in *density()*
- Updated *logo example*

pyrolite.util

- Updated *pyrolite.util.meta*
- Added `pyrolite.util.plot.conditional_prob_density()`; added conditional `statsmodels` import within *pyrolite.util.plot* to access `KDEMultivariateConditional`
- Added keyword argument `logy` to *pyrolite.util.math.interpolate_line()*
- Added *pyrolite.util.math.grid_from_ranges()* and *pyrolite.util.math.flattengrid()*
- Added support for differential x-y padding in `pyrolite.util.plot.get_full_extent()` and `pyrolite.util.plot.save_axes()`
- Added *pyrolite.util.skl.pipeline.fit_save_classifier()* (then `pyrolite.util.skl.fit_save_classifier`)

5.1.38 0.1.4

pyrolite.plot

- Updated relevant docs and references for *pyrolite.plot* and the *pyrolite.plot.pyroplot* DataFrame accessor

pyrolite.comp

- Expanded *pyrolite.comp.impute* and improved *pyrolite.comp.impute.EMCOMP()*
- Added EMCOMP example (later removed in 0.2.5, pending validation and improvements for EMCOMP).

pyrolite.util

- Updated *pyrolite.util.meta* with docstring utilities `numpydoc_str_param_list()` and `get_additional_params()`

5.1.39 0.1.2

- Fixed logo naming issue in docs

pyrolite.plot

- Bugfixes for *pyrolite.plot.density.density()* (then `pyrolite.plot.density`) and `pyrolite.plot.util.ternary_heatmap()`

5.1.40 0.1.1

pyrolite.plot

- Added *logo example*
- Refactored *pyrolite.plot* to use the *pyrolite.plot.pyroplot* DataFrame accessor:
  - Renamed `pyrolite.plot.spiderplot` to *pyrolite.plot.spider.spider()*
  - Renamed `pyrolite.plot.spider.REE_radii_plot` to *pyrolite.plot.spider.REE_v_radii()*
  - Renamed `pyrolite.plot.ternaryplot` to `pyrolite.plot.tern.ternary()`
  - Renamed `pyrolite.plot.densityplot` to *pyrolite.plot.density.density()*
- Updated *pyrolite.plot.density.density()* and `pyrolite.plot.tern.ternary()`

pyrolite.comp

- Bugfixes and improvements for *pyrolite.comp.impute*

pyrolite.geochem

- Updated *oxide_conversion()* and *convert_chemistry()*

pyrolite.util

- Added `plot_stdev_ellipses()` and `plot_pca_vectors()`
- Updated `pyrolite.util.plot.plot_Z_percentiles()`
- Updated `pyrolite.util.plot.ternary_heatmap()`
- Updated `pyrolite.util.plot.vector_to_line()`

5.1.41 0.1.0

pyrolite.plot

- Updates to *pyrolite.plot.density.density()* to better deal with linear/log spaced and a ternary heatmap

pyrolite.comp

- Added *EMCOMP()* to *pyrolite.comp.impute*
- Renamed `inv_alr`, `inv_clr`, `inv_ilr` and `inv_boxcox` to `inverse_alr()`, `inverse_clr()`, `inverse_ilr()` and *inverse_boxcox()*

pyrolite.util

- Added *pyrolite.util.synthetic*
- Moved `pyrolite.util.pd.normal_frame` and `pyrolite.util.pd.normal_series` to *pyrolite.util.synthetic.normal_frame()* and *pyrolite.util.synthetic.normal_series()*
- Added *pyrolite.util.missing* and *pyrolite.util.missing.md_pattern()*

- Added *pyrolite.util.math.eigsorted()*, *pyrolite.util.math.augmented_covariance_matrix()*, *pyrolite.util.math.interpolate_line()*

Note

Releases before 0.1.0 are available via GitHub for reference, but were `alpha` versions which were never considered stable.

5.2 Roadmap

This page details some of the under-development and planned features for `pyrolite`. Note that while no schedules are attached, features under development are likely to be completed with weeks to months, while those 'On The Horizon' may be significantly further away (or in some cases may not make it to release).

5.2.1 Under Development

These features are either under development or planned to be implemented and should be released in the near future.

- **Interactive Plotting Backend Options**: `pyrolite` visualisation is currently based entirely on static plot generation via `matplotlib`. While this works well for publication-style figures, it may be possible to leverage `pandas`-based frameworks to provide options for alternative backends, some of which are more interactive and amendable to data exploration (e.g. `hvplot`, `plotly`). See the `pandas` extension docs for one option for implementing this (plotting-backends).
- **Units and Uncertainty Aware Geochemistry DataFrames**: Work has started on a prototype implementation to bring in native data types related to units, and incorporate uncertainties which can be propogated through calculations. This is principally based around `pint`, `pint_pandas` and `uncertainties`.

5.2.2 On the Horizon, Potential Future Updates

These are a number of features which are in various stages of development, which are planned be integrated over the longer term.

pyrolite.mineral

There are a few components which will make better use of mineral chemistry data, and facilitate better comparison of whole-rock and mineral data (Issue #5):

- Normative mineral calculations
- Mineral formulae recalculation, site partitioning, endmember calculations

pyrolite.geochem.isotope

- Stable isotope calculations
- Simple radiogenic isotope system calculations and plots

pyrolite.comp.impute

Expansion of compositional data imputation algorithms beyond EMCOMP (Issue #6).

pyrolite.geochem.magma

Utilities for simple melting and fractionation models.

`pyrolite.geochem.quality`

Utilities for: : - assessing data quality

- identifying potential analytical artefacts
- assessing uncertainties

5.2.3 Governance and Documentation

- Depending on how the community grows, and whether `pyrolite` brings with it a series of related tools, the project and related tools may be migrated to an umbrella organization on GitHub (e.g. `pyrolite/pyrolite`) so they can be collectively managed by a community.
- **Internationalization**: While the pyrolite source is documented in English, it would be good to be able to provide translated versions of the documentation to minimise hurdles to getting started.
- **Teaching Resources**: `pyrolite` is well placed to provide solutions and resources for use in under/post-graduate education. While we have documentation sections dedicated to examples and tutorials, perhaps we could develop explicit sections for educational resources and exercises.

5.3 Code of Conduct

5.3.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

5.3.2 Our Standards

Examples of behaviour that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behaviour by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.3.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behaviour and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behaviour.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviours that they deem inappropriate, threatening, offensive, or harmful.

5.3.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

5.3.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behaviour may be reported by contacting the project admins. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

5.3.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant Version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html. The Contributor Covenant is released under the Creative Commons Attribution 4.0 License.

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq.

5.4 Contributing

The long-term aim of this project is to be designed, built and supported by (and for) the geochemistry community. In the present, the majority of the work involves incorporating geological knowledge and frameworks into a practically useful core set of tools which can be later be expanded. As such, requests for features and bug reports are particularly valuable contributions, in addition to code and expanding the documentation. All individuals contributing to the project are expected to follow the *Code of Conduct*, which outlines community expectations and responsibilities.

Also, be sure to add your name or GitHub username to the *contributors list*.

Note

This project is currently in `beta`, and as such there's much work to be done.

5.4.1 Feature Requests

If you're new to Python, and want to implement a specific process, plot or framework as part of `pyrolite`, you can submit a Feature Request. Perhaps also check the Issues Board first to see if someone else has suggested something similar (or if something is in development), and comment there.

5.4.2 Bug Reports

If you've tried to do something with `pyrolite`, but it didn't work, and googling error messages didn't help (or, if the error messages are full of `pyrolite.XX.xx`), you can submit a Bug Report . Perhaps also check the Issues Board first to see if someone else is having the same issue, and comment there.

5.4.3 Contributing to Documentation

The documentation and examples for `pyrolite` are gradually being developed, and any contributions or corrections would be greatly appreciated. Currently the examples are patchy, and any 'getting started' guides would be a helpful addition.

These pages serve multiple purposes:

- A human-readable reference of the source code (compiled from docstrings).
- A set of simple examples to demonstrate use and utility.
- A place for developing extended examples[1]

5.4.4 Contributing Code

Code contributions are always welcome, whether it be small modifications or entire features. As the project gains momentum, check the Issues Board for outstanding issues, features under development. If you'd like to contribute, but you're not so experienced with Python, look for `good first issue` tags or email the maintainer for suggestions.

To contribute code, the place to start will be forking the source for `pyrolite` from GitHub. Once forked, clone a local copy and from the repository directory you can install a development (editable) copy via `python setup.py develop`. To incorporate suggested changes back to into the project, push your changes to your remote fork, and then submit a pull request onto pyrolite/develop or a relevant feature branch.

Note

- See *Development* for directions for installing extra dependencies for development and for information on development environments and tests.
- `pyrolite` development roughly follows a gitflow workflow. `pyrolite/main` is only used for releases, and large separable features should be build on `feature` branches off `develop`.
- Contributions introducing new functions, classes or entire features should also include appropriate tests where possible (see [Writing Tests], below).
- `pyrolite` uses Black for code formatting, and submissions which have passed through `Black` are appreciated, although not critical.

5.4.5 Writing Tests

There is currently a broad unit test suite for `pyrolite`, which guards against breaking changes and assures baseline functionality. `pyrolite` uses continuous integration via GitHub Actions, where the full suite of tests are run for each commit and pull request, and test coverage output to Coveralls.

Adding or expanding tests is a helpful way to ensure `pyrolite` does what is meant to, and does it reproducibly. The unit test suite one critical component of the package, and necessary to enable sufficient trust to use `pyrolite` for scientific purposes.

5.5 Contributors

This list includes people who have contributed to the project in the form of code, comments, testing, bug reports, or feature requests.

- Morgan Williams

[1] Such examples could easily be distributed as educational resources showcasing the utility of programmatic approaches to geochemistry

- Hayden Dalton
- Louise Schoneveld
- Adam Bath
- Yajing Mao
- Justin Gosses
- Kaarel Mand
- Laura Miller
- Steve Barnes
- Lucy Mathieson
- Nicolas Piette-Lauziere
- Chetan Nathwani
- Martin Bentley
- Tom Buckle
- Angela Rodrigues
- Sarah Shi
- Ondrej Lexa
- Bob Myhill
- Malte Mues
- Jiawei Yun
- Jarrod Burges

5.6 Development

5.6.1 Development History and Planning

- *Changelog*
- *Roadmap*

5.6.2 Contributing

- *Contributing*
- *Contributors*
- *Code of Conduct*

5.6.3 Development Installation

To access and use the development version, you can clone the repository and set up the environment:

```
git clone https://github.com/morganjwilliams/pyrolite.git
git checkout develop
uv sync --extra dev
```

5.6.4 Tests

If you clone the source repository, unit tests can be run using pytest from the root directory after installation with development dependencies (`pip install -e .[dev]`):

```
uv run pytest
```

If instead you only want to test a subset, you can call `pytest` directly from within the pyrolite repository:

```
uv run pytest ./test/<path to test or test folder>
```

5.7 Release Guide

> **Note**
>
> There is now a 'publish' action in the repository which handles releases.

Before releasing, ensure that the changelog is up to date, consider preloading the next release link, and ensure the `uv` lock is up to date:

```
uv sync --extra dev
git add uv.lock
git commit "Update uv Lock"
```

The project can be built into distributable artefacts (note that `build`, `dist` and `pyrolite.egg-info` directories should be removed first) with:

```
uv run python -m build --sdist --wheel
```

We can check this distribution with `twine`:

```
uv run twine check ./dist/*
```

And, assuming credentials exsit, uploaded to PyPI with:

```
uv run twine upload ./dist/*
```

CHAPTER

SIX

REFERENCE

6.1 API

6.1.1 pyrolite.plot

Submodule with various plotting and visualisation functions.

pyrolite.plot.pyroplot (Pandas Interface)

class pyrolite.plot.**pyroplot**(*obj*)

cooccurence(*ax: Axes | None = None*, *normalize: bool = True*, *log: bool = False*, *colorbar: bool = False*, ***kwargs*) → Axes

Plot the co-occurence frequency matrix for a given input.

Parameters

- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **normalize** (*bool*) – Whether to normalize the cooccurence to compare disparate variables.
- **log** (*bool*) – Whether to take the log of the cooccurence.
- **colorbar** (*bool*) – Whether to append a colorbar.

Returns
Axes on which the cooccurence plot is added.

Return type
matplotlib.axes.Axes

density(*components: list[str] | None = None*, *ax: Axes | None = None*, *axlabels: bool = True*, ***kwargs*) → Axes

Method for plotting histograms (mode='hist2d'|'hexbin') or kernel density esitimates from point data. Convenience access function to *~pyrolite.plot.density.density*, where further parameters for relevant *matplotlib* functions are also listed.

Parameters

- **components** (*list*) – Elements or compositional components to plot.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **axlabels** (*bool*) – Whether to add x-y axis labels.

Returns
Axes on which the density diagram is plotted.

Return type
matplotlib.axes.Axes

Notes

See also: `pyrolite.plot.pyroplot.density()`, `pyrolite.plot.density.density()`.

heatscatter(*components: list[str] | None = None*, *ax: Axes | None = None*, *axlabels: bool = True*, *logx: bool = False*, *logy: bool = False*, ***kwargs*) → Axes

Heatmapped scatter plots using the pyroplot API. See further parameters for *matplotlib.pyplot.scatter* function below.

Parameters

- **components** (*list*) – Elements or compositional components to plot.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **axlabels** (*bool*) – Whether to add x-y axis labels.
- **logx** (*bool*) – Whether to log-transform x values before the KDE for bivariate plots.
- **logy** (*bool*) – Whether to log-transform y values before the KDE for bivariate plots.

Returns
Axes on which the heatmapped scatterplot is added.

Return type
matplotlib.axes.Axes

Notes

See also: `pyrolite.plot.pyroplot.scatter()`.

parallel(*components: list[str] | None = None*, *rescale: bool = False*, *legend: bool = False*, *ax: Axes | None = None*, ***kwargs*) → Axes

Create a *pyrolite.plot.parallel.parallel*. coordinate plot from the columns of the pandas.DataFrame.

Parameters

- **components** (*list*) – Components to use as axes for the plot.
- **rescale** (*bool*) – Whether to rescale values to [-1, 1].
- **legend** (*bool*) – Whether to include or suppress the legend.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.

Returns
Axes on which the parallel coordinates plot is added.

Return type
matplotlib.axes.Axes

Todo

- Adapt figure size based on number of columns.

Notes

See also: *pyrolite.plot.pyroplot.parallel*, *pyrolite.plot.parallel.parallel*.

plot(*components: list[str] | None = None*, *ax: Axes | None = None*, *axlabels: bool = True*, ***kwargs*) → Axes

Convenience method for line plots using the pyroplot API. See further parameters for *matplotlib.pyplot.scatter* function below.

Parameters

- **components** (*list*) – Elements or compositional components to plot.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **axlabels** (*bool*) – Whether to add x-y axis labels.

Returns
Axes on which the plot is added.

Return type
matplotlib.axes.Axes

Notes

See also: `pyrolite.plot.pyroplot.plot()`, `matplotlib.pyplot.plot()`

REE(*index: str = 'elements'*, *ax: Axes | None = None*, *mode: str = 'plot'*, *dropPm: bool = True*, *scatter_kw: dict | None = None*, *line_kw: dict | None = None*, ***kwargs*) → Axes

Pass the pandas object to `pyrolite.plot.spider.REE_v_radii()`.

Parameters

- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **index** (*str*) – Whether to plot radii ('radii') on the principal x-axis, or elements ('elements').
- **mode** (*str*) – Mode for plot. Plot will produce a line-scatter diagram. Fill will return a filled range. Density will return a conditional density diagram.
- **dropPm** (*bool*) – Whether to exclude the (almost) non-existent element Promethium from the REE list.
- **scatter_kw** (*dict*) – Keyword parameters to be passed to the scatter plotting function.
- **line_kw** (*dict*) – Keyword parameters to be passed to the line plotting function.

Returns
Axes on which the REE plot is added.

Return type
matplotlib.axes.Axes

Notes

See also: `pyrolite.plot.pyroplot.REE()`, `pyrolite.plot.spider.REE_v_radii()`.

scatter(*components: list[str] | None = None*, *ax: Axes | None = None*, *axlabels: bool = True*, ***kwargs*) → Axes

Convenience method for scatter plots using the pyroplot API. See further parameters for *matplotlib.pyplot.scatter* function below.

Parameters

- **components** (*list*) – Elements or compositional components to plot.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **axlabels** (*bool*) – Whether to add x-y axis labels.

Returns
Axes on which the scatterplot is added.

Return type
matplotlib.axes.Axes

Notes

See also: `pyrolite.plot.pyroplot.heatscatter()`, `matplotlib.pyplot.scatter()`

spider(*components: list[str] | None = None*, *indexes: list[float] | None = None*, *ax: Axes | None = None*, *mode: str = 'plot'*, *index_order: str | list[str] | None = None*, *autoscale: bool = True*, *scatter_kw: dict | None = None*, *line_kw: dict | None = None*, ***kwargs*) → Axes

Method for spider plots. Convenience access function to :func"~*pyrolite.plot.spider.spider*, where further parameters for relevant *matplotlib* functions are also listed.

Parameters

- **components** (*list*) – Elements or compositional components to plot.
- **indexes** (*list*) – Positions of the elements along the xaxis.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **index_order** – Function to order spider plot indexes (e.g. by incompatibility).
- **autoscale** (*bool*) – Whether to autoscale the y-axis limits for standard spider plots.
- **mode** (*str*) – Mode for plot. Plot will produce a line-scatter diagram. Fill will return a filled range. Density will return a conditional density diagram.
- **scatter_kw** (*dict*) – Keyword parameters to be passed to the scatter plotting function.
- **line_kw** (*dict*) – Keyword parameters to be passed to the line plotting function.

Returns
Axes on which the spider diagram is plotted.

Return type
matplotlib.axes.Axes

Notes

See also: `pyrolite.plot.pyroplot.spider()`, `pyrolite.plot.spider.spider()`.

stem(*components: list[str] | None = None*, *ax: Axes | None = None*, *orientation: str = 'horizontal'*, *axlabels: bool = True*, ***kwargs*) → Axes

Method for creating stem plots. Convenience access function to `stem()`, where further parameters for relevant *matplotlib* functions are also listed.

Parameters

- **components** (*list*) – Elements or compositional components to plot.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **orientation** (*str*) – Orientation of the plot (horizontal or vertical).
- **axlabels** (*bool*) – Whether to add x-y axis labels.

Returns
Axes on which the stem diagram is plotted.

Return type
matplotlib.axes.Axes

Notes

See also: `pyrolite.plot.pyroplot.stem()`,:func:*pyrolite.plot.stem.stem*.

pyrolite.plot.spider

pyrolite.plot.spider.**spider**(*arr: ndarray*, *indexes=None*, *ax: Axes | None = None*, *label: str | None = None*, *logy: bool = True*, *yextent: tuple[float, float] | None = None*, *mode: str = 'plot'*, *unity_line: bool = False*, *scatter_kw: dict | None = None*, *line_kw: dict | None = None*, *set_ticks: bool = True*, *autoscale: bool = True*, ***kwargs*) → Axes

Plots spidergrams for trace elements data. Additional arguments are typically forwarded to respective `matplotlib` functions `plot()` and `scatter()`.

Parameters

- **arr** (*numpy.ndarray*) – Data array.
- **indexes** (*: numpy.ndarray*) – Numerical indexes of x-axis positions.
- **ax** (matplotlib.axes.Axes, *None*) – The subplot to draw on.
- **label** (str, *None*) – Label for the individual series.
- **logy** (*bool*) – Whether to use a log y-axis.
- **yextent** (*tuple*) – Extent in the y direction for conditional probability plots, to limit the gridspace over which the kernel density estimates are evaluated.
- **mode** (str, *["plot", "fill", "binkde", "ckde", "kde", "hist"]*) – Mode for plot. Plot will produce a line-scatter diagram. Fill will return a filled range. Density will return a conditional density diagram.
- **unity_line** (*bool*) – Add a line at y=1 for reference.
- **scatter_kw** (*dict*) – Keyword parameters to be passed to the scatter plotting function.
- **line_kw** (*dict*) – Keyword parameters to be passed to the line plotting function.
- **set_ticks** (*bool*) – Whether to set the x-axis ticks according to the specified index.
- **autoscale** (*bool*) – Whether to autoscale the y-axis limits for standard spider plots.

Returns
Axes on which the spiderplot is plotted.

Return type
matplotlib.axes.Axes

Notes

By using separate lines and scatterplots, values between two missing items are still presented.

See also: * `matplotlib.pyplot.plot()` * `matplotlib.pyplot.scatter()` * `REE_v_radii()`.

Todo

- Might be able to speed up lines with `LineCollection`.
- Legend entries

pyrolite.plot.spider.**REE_v_radii**(*arr: ndarray | None = None*, *ax: Axes | None = None*, *ree: list[str] | None = None*, *index: str = 'elements'*, *mode: str = 'plot'*, *logy: bool = True*, *tl_rotation: float = 60*, *unity_line: bool = False*, *scatter_kw: dict | None = None*, *line_kw: dict | None = None*, *set_labels: bool = True*, *set_ticks: bool = True*, ***kwargs*) → Axes

Creates an axis for a REE diagram with ionic radii along the x axis.

Parameters

- **arr** (*numpy.ndarray*) – Data array.
- **ax** (matplotlib.axes.Axes, *None*) – Optional designation of axes to reconfigure.
- **ree** (*list*) – List of REE to use as an index.
- **index** (*str*) – Whether to plot using radii on the x-axis ('radii'), or elements ('elements').
- **mode** (str, *["plot", "fill", "binkde", "ckde", "kde", "hist"]*) – Mode for plot. Plot will produce a line-scatter diagram. Fill will return a filled range. Density will return a conditional density diagram.
- **logy** (*bool*) – Whether to use a log y-axis.
- **tl_rotation** (*float*) – Rotation of the numerical index labels in degrees.
- **unity_line** (*bool*) – Add a line at y=1 for reference.
- **scatter_kw** (*dict*) – Keyword parameters to be passed to the scatter plotting function.
- **line_kw** (*dict*) – Keyword parameters to be passed to the line plotting function.
- **set_labels** (*bool*) – Whether to set the x-axis ticklabels for the REE.
- **set_ticks** (*bool*) – Whether to set the x-axis ticks according to the specified index.

Returns
Axes on which the REE_v_radii plot is added.

Return type
matplotlib.axes.Axes

Todo

- Turn this into a plot template within pyrolite.plot.templates submodule

Notes

See also:

- *matplotlib.pyplot.plot*
- *matplotlib.pyplot.scatter*
- *~pyrolite.plot.spider.spider*

- *pyrolite.geochem.transform.lambda_lnREE*

pyrolite.plot.density

Kernel desnity estimation plots for geochemical data.

pyrolite.plot.density.**density**(*arr: ndarray*, *ax: Axes | None = None*, *logx: bool = False*, *logy: bool = False*, *bins: int = 25*, *mode: str = 'density'*, *extent: tuple[float] | None = None*, *contours: list[float] | None = None*, *percentiles: bool = True*, *relim: bool = True*, *cmap: str | Colormap | None = <matplotlib.colors.ListedColormap object>*, *shading: str = 'auto'*, *vmin: float = 0.0*, *colorbar: bool = False*, ***kwargs*) → Axes

Creates diagramatic representation of data density and/or frequency for either binary diagrams (X-Y) or ternary plots. Additional arguments are typically forwarded to respective `matplotlib` functions `pcolormesh()`, `hist2d()`, `hexbin()`, `contour()`, and `contourf()`.

Parameters

- **arr** (*numpy.ndarray*) – Dataframe from which to draw data.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **logx** (*bool*) – Whether to use a logspaced *grid* on the x axis. Values strictly >0 required.
- **logy** (*bool*) – Whether to use a logspaced *grid* on the y axis. Values strictly >0 required.
- **bins** (*int, 20*) – Number of bins used in the gridded functions (histograms, KDE evaluation grid).
- **mode** (*str*) – Different modes used here: [‘density’, ‘hexbin’, ‘hist2d’]
- **extent** (*list*) – Predetermined extent of the grid for which to from the histogram/KDE. In the general form (xmin, xmax, ymin, ymax).
- **contours** (*list*) – Contours to add to the plot, where *mode=’density’* is used.
- **percentiles** (*bool*) – Whether contours specified are to be converted to percentiles.
- **relim** (*bool*) – Whether to relimit the plot based on xmin, xmax values.
- **cmap** (*matplotlib.colors.Colormap*) – Colormap for mapping surfaces.
- **vmin** (*float*) – Minimum value for colormap.
- **shading** (*str*) – Shading to apply to pcolormesh.
- **colorbar** (*bool*) – Whether to append a linked colorbar to the generated mappable image.

Returns
Axes on which the densityplot is plotted.

Return type
matplotlib.axes.Axes

Notes

The default density estimates and derived contours are generated based on kernel density estimates. Assumptions around e.g. 95% of points lying within a 95% contour won’t necessarily be valid for non-normally distributed data (instead, this represents the approximate 95% percentile on the kernel density estimate). Note that contours are currently only generated; for *mode=”density”*; future updates may allow the use of a histogram basis, which would give results closer to 95% data percentiles.

See also: * matplotlib.pyplot.pcolormesh() * matplotlib.pyplot.hist2d() * matplotlib.pyplot.contourf() * matplotlib.pyplot.hexbin() * matplotlib.pyplot.contour() * matplotlib.pyplot.contourf()

Todo

- Allow generation of contours from histogram data, rather than just the kernel density estimate.
- Implement an option and filter to 'scatter' points below the minimum threshold or maximum percentile contours.

pyrolite.plot.density.grid

class pyrolite.plot.density.grid.**DensityGrid**(*x: ndarray*, *y: ndarray*, *extent: tuple[float] | None = None*, *bins: int = 50*, *logx: bool = False*, *logy: bool = False*, *coverage_scale: float = 1.2*)

calculate_grid()

get_ystep() → float

get_xstep() → float

extent_from_xy(*x: ndarray*, *y: ndarray*, *coverage_scale: float | None = None*) → list[tuple[float, float]]

get_xrange() → tuple[float, float]

get_yrange() → tuple[float, float]

get_extent() → tuple[float, float, float, float]

get_range() → list[list[float]]

update_grid_centre_ticks()

update_grid_edge_ticks()

get_centre_grid()

get_edge_grid()

get_hex_extent() → list[float]

kdefrom(*xy: ndarray*, *xtransform: Callable = <function DensityGrid.<lambda>>*, *ytransform: Callable = <function DensityGrid.<lambda>>*, *mode: str = 'centres'*, *bw_method: str | float | Callable | None = None*)

Take an x-y array and sample a KDE on the grid.

pyrolite.plot.density.ternary

pyrolite.plot.density.ternary.**ternary_heatmap**(*data: ndarray*, *bins: int = 20*, *mode: str = 'density'*, *transform: Callable = <function ILR>*, *inverse_transform: Callable = <function inverse_ILR>*, *ternary_min_value: float = 0.0001*, *grid_border_frac: float = 0.1*, *grid: ndarray | None = None*, ***kwargs*) → tuple[ndarray, ndarray, ndarray, ndarray, dict[str, ndarray | Callable]]

Heatmap for ternary diagrams. This invokes a 3D to 2D transform such as a log transform prior to creating a grid.

Parameters

- **data** (*numpy.ndarray*) – Ternary data to obtain heatmap coords from.
- **bins** (*int*) – Number of bins for the grid.
- **mode** (str, *{'histogram', 'density'}*) – Which mode to render the histogram/KDE in.
- **transform** (*Callable | sklearn.base.TransformerMixin*) – Callable function or Transformer class.
- **inverse_transform** (*Callable*) – Inverse function for *transform*, necessary if transformer class not specified.
- **ternary_min_value** (*float*) – Optional specification of minimum values within a ternary diagram to draw the transformed grid.
- **grid_border_frac** (*float*) – Size of border around the grid, expressed as a fraction of the total grid range.
- **grid** (*numpy.ndarray*) – Grid coordinates to sample at, if already calculated. For the density mode, this is a (nsamples, 2) array. For histograms, this is a two-member list of bin edges.

Returns

- **t, l, r** (*tuple of numpy.ndarray*) – Ternary coordinates for the heatmap.
- **H** (*numpy.ndarray*) – Histogram/density estimates for the coordinates.
- **data** (*dict*) – Data dictonary with grid arrays and relevant information.

Notes

Zeros will not render in this heatmap, consider replacing zeros with small values or imputing them if they must be incorporated.

pyrolite.plot.parallel

pyrolite.plot.parallel.**parallel**(*df: DataFrame*, *components: list[str] | None = None*, *rescale: bool = True*, *legend: bool = False*, *ax: Axes | None = None*, *label_rotate: float = 60*, ***kwargs*) → Axes

Create a parallel coordinate plot across dataframe columns, with individual lines for each row.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to create a plot from.
- **components** (*list*) – Subset of dataframe columns to use as indexes along the x-axis.
- **rescale** (*bool*) – Whether to rescale values to [-1, 1].
- **legend** (*bool*) – Whether to include or suppress the legend.
- **ax** (*matplotlib.axes.Axes*) – Axis to plot on (optional).

Return type
matplotlib.axes.Axes

Todo

- A multi-axis version would be more compatible with independent rescaling and zoom
- Enable passing a list of colors Rather than just a list of numbers to be converted to colors.

pyrolite.plot.stem

pyrolite.plot.stem.**stem**(*x: ndarray[tuple[int], dtype[number]] | Series*, *y: ndarray[tuple[int], dtype[number]] | Series*, *ax: Axes | None = None*, *orientation: str = 'horizontal'*, ***kwargs*) → Axes

Create a stem (or 'lollipop') plot, with optional orientation.

Parameters

- **x, y** (*numpy.ndarray*) – 1D arrays for independent and dependent axes.
- **ax** (*matplotlib.axes.Axes*) – The subplot to draw on.
- **orientation** (*str*) – Orientation of the plot (horizontal or vertical).

Returns
Axes on which the stem diagram is plotted.

Return type
matplotlib.axes.Axes

pyrolite.plot.biplot

pyrolite.plot.biplot.**compositional_SVD**(*X: ndarray*) → tuple[ndarray, ndarray]

Breakdown a set of compositions to vertexes and cases for adding to a compositional biplot.

Parameters
X (*numpy.ndarray*) – Compositional array.

Returns
vertexes, cases

Return type
numpy.ndarray, numpy.ndarray

pyrolite.plot.biplot.**plot_origin_to_points**(*xs: ndarray*, *ys: ndarray*, *labels: list[str] | None = None*, *ax: Axes | None = None*, *origin: tuple[float, float] = (0.0, 0.0)*, *color: str = 'k'*, *marker: str = 'o'*, *pad: float = 0.05*, ***kwargs*) → Axes

Plot lines radiating from a specific origin. Fornulated for creation of biplots (*covariance_biplot*, *compositional_biplot*).

Parameters

- **xs, ys** (*numpy.ndarray*) – Coordinates for points to add.
- **labels** (*list*) – Labels for verticies.
- **ax** (*matplotlib.axes.Axes*) – Axes to plot on.
- **origin** (*tuple*) – Origin to plot from.
- **color** (*str*) – Line color to use.
- **marker** (*str*) – Marker to use for ends of vectors and origin.
- **pad** (*float*) – Fraction of vector to pad text label.

Returns
Axes on which radial plot is added.

Return type
matplotlib.axes.Axes

pyrolite.plot.biplot.**compositional_biplot**(*data: ndarray*, *labels: list[str] | None = None*, *ax: Axes | None = None*, ***kwargs*) → Axes

Create a compositional biplot.

Parameters

- **data** (*numpy.ndarray*) – Coordinates for points to add.
- **labels** (*list*) – Labels for verticies.
- **ax** (*matplotlib.axes.Axes*) – Axes to plot on.

Returns
Axes on which biplot is added.

Return type
matplotlib.axes.Axes

pyrolite.plot.templates

A utility submodule for standardised plot templates to be added to matplotlib axes.

Todo

- Make use of new ax.axline features (https://matplotlib.org/3.3.1/users/whats_new.html#new-axes-axline-method)

pyrolite.plot.templates.**QAP**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *color: str = 'k'*, ***kwargs*) → Axes

IUGS QAP ternary classification diagram **[#pyrolite.plot.templates.QAP.QAP_1]_** **[#pyrolite.plot.templates.QAP.QAP_2]_**.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Ternary axes to add the diagram to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field ‘name’ or ‘ID’.
- **color** (*str*) – Color for the polygon edges in the diagram.

References

pyrolite.plot.templates.**TAS**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *relim: bool = True*, *color: str = 'k'*, *which_model: str | None = None*, ***kwargs*) → Axes

Adds the TAS diagram to an axes. Diagram from Middlemost (1994) **[#pyrolite.plot.templates.TAS.TAS_1]_**, a closed-polygon variant after Le Bas et al (1992) **[#pyrolite.plot.templates.TAS.TAS_2]_**.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the template on to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.

- **which_labels** (*str*) – Which labels to add to the polygons (e.g. for TAS, 'volcanic', 'intrusive' or the field 'ID').
- **relim** (*bool*) – Whether to relimit axes to fit the built in ranges for this diagram.
- **color** (*str*) – Line color for the diagram.
- **which_model** (*str*) – The name of the model variant to use, if not Middlemost.

Returns
ax

Return type
matplotlib.axes.Axes

References

pyrolite.plot.templates.**FeldsparTernary**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *mode: str = 'miscibility-gap'*, *color: str = 'k'*, ***kwargs*) → Axes

Simplified feldspar classifcation diagram, based on a version printed in the second edition of 'An Introduction to the Rock Forming Minerals' (Deer, Howie and Zussman). [#pyrolite.plot.templates.feldspar.FeldsparTernary_1]

Parameters

- **ax** (*matplotlib.axes.Axes*) – Ternary axes to add the diagram to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **mode** (*str*) – Which mode of the diagram to use; the two implemented for the feldspar ternary diagram are 'default' and 'miscibility-gap', the second of which provides a simplified approximation of the miscibility gap between k-feldspar and plagioclase.
- **color** (*str*) – Color for the polygon edges in the diagram.

References

pyrolite.plot.templates.**Herron**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *relim: bool = True*, *color: str = 'k'*, ***kwargs*) → Axes

Adds the Herron (1988) [#pyrolite.plot.templates.sandstones.Herron_1] sandstones classification diagram.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the template on to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **relim** (*bool*) – Whether to relimit axes to fit the built in ranges for this diagram.
- **color** (*str*) – Line color for the diagram.

Returns
ax

Return type
matplotlib.axes.Axes

References

pyrolite.plot.templates.**JensenPlot**(*ax: Axes | None = None*, *add_labels: bool = False*, *color: str = 'k'*, ***kwargs*) → Axes

Jensen Plot for classification of sub-alkaline volcanic rocks **[#pyrolite.plot.templates.jensen.JensenPlot_1]_**.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the template on to.
- **add_labels** (*bool*) – Whether to include the labels for the diagram.
- **color** (*str*) – Line color for the diagram.

Returns
ax

Return type
matplotlib.axes.Axes

References

Notes

Diagram used for the classification classification of subalkalic volcanic rocks. The diagram is constructed for molar cation percentages of Al, Fe+Ti and Mg, on account of these elements' stability upon metamorphism. This particular version uses updated labels relative to Jensen (1976), in which the fields have been extended to the full range of the ternary plot.

pyrolite.plot.templates.**Pettijohn**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *relim: bool = True*, *color: str = 'k'*, ***kwargs*) → Axes

Adds the Pettijohn (1973) [#pyrolite.plot.templates.sandstones.Pettijohn_1] sandstones classification diagram.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the template on to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **relim** (*bool*) – Whether to relimit axes to fit the built in ranges for this diagram.
- **color** (*str*) – Line color for the diagram.

Returns
ax

Return type
matplotlib.axes.Axes

References

pyrolite.plot.templates.**SpinelFeBivariate**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *color: str = 'k'*, ***kwargs*) → Axes

Fe-Spinel classification, designed for data in atoms per formula unit.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the diagram to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.

- **color** (*str*) – Color for the polygon edges in the diagram.

pyrolite.plot.templates.**SpinelTrivalentTernary**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *color: str = 'k'*, ***kwargs*) → Axes

Spinel Trivalent Ternary classification - designed for data in atoms per formula unit.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Ternary axes to add the diagram to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **color** (*str*) – Color for the polygon edges in the diagram.

pyrolite.plot.templates.**USDASoilTexture**(*ax: Axes | None = None*, *add_labels: bool = False*, *color: str = 'k'*, ***kwargs*) → Axes

United States Department of Agriculture Soil Texture classification model **[#pyrolite.plot.templates.USDA_soil_texture.USDASoilTexture_1]_** **[#pyrolite.plot.templates.USDA_soil_texture.USDASoilTexture_2]_**.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the template on to.
- **add_labels** (*bool*) – Whether to include the labels for the diagram.
- **color** (*str*) – Line color for the diagram.

Returns
ax

Return type
matplotlib.axes.Axes

References

pyrolite.plot.templates.**pearceThNbYb**(*ax: Axes | None = None*, *relim: bool = True*, *color: str = 'k'*, ***kwargs*) → Axes

Adds the Th-Nb-Yb delimiter lines from Pearce (2008) **[#pyrolite.plot.templates.pearce.pearceThNbYb_1]_** to an axes.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the template onto.
- **relim** (*bool*) – Whether to relimit axes to fit the built in ranges for this diagram.
- **color** (*str*) – Line color for the diagram.

References

Returns
ax

Return type
matplotlib.axes.Axes

pyrolite.plot.templates.**pearceTiNbYb**(*ax: Axes | None = None*, *relim: bool = True*, *color: str = 'k'*, *annotate: bool = True*, ***kwargs*) → Axes

Adds the Ti-Nb-Yb delimiter lines from Pearce (2008) **[#pyrolite.plot.templates.pearce.pearceTiNbYb_1]_** to an axes.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the template onto.
- **relim** (*bool*) – Whether to relimit axes to fit the built in ranges for this diagram.
- **color** (*str*) – Line color for the diagram.

References

Returns
ax

Return type
matplotlib.axes.Axes

pyrolite.plot.templates.**Magnetite**(*ax: Axes | None = None*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_model: str | None = None*, *relim: bool = True*, *color: str = 'k'*, ***kwargs*) → Axes

Magnetite trace element discrimination diagram for mineral deposit types from Dupuis and Beaudoin (2011) **[#pyrolite.plot.templates.magnetite.Magnetite_1]_**.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to add the diagram to.
- **add_labels** (*bool*) – Whether to add labels at polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_model** (*str*) – Diagram to use: 'TiV_CaAlMn' (Ca+Al+Mn vs Ti+V; default) or 'TiV_NiCrMn' (Ni/(Cr+Mn) vs Ti+V).
- **relim** (*bool*) – Whether to relimit axes to the default range for this diagram.
- **color** (*str*) – Color for the polygon edges in the diagram.

References

pyrolite.plot.color

pyrolite.plot.color.**get_cmode**(*c: str | list | tuple | ndarray | None = None*) → str

Find which mode a color is supplied as, such that it can be processed.

Parameters
c (*str | list | tuple | numpy.ndarray*) – Color arguments as typically passed to *matplotlib.pyplot.scatter* or *matplotlib.pyplot.plot*.

pyrolite.plot.color.**process_color**(*c: str | list | tuple | ndarray | None = None*, *color: str | tuple[float] | list[str] | list[tuple[float, ...]] | ~numpy.ndarray | None=None*, *cmap: str | Colormap | None = None*, *alpha: float | None = None*, *norm: Norm | None = None*, *bad: str | tuple[float] = '0.5'*, *cmap_under: str | tuple[float, ...]=(1.0, 1.0, 1.0, 0.0)*, *color_converter: Callable = <function to_rgba>*, *color_mappings: dict | None = None*, *size: int | None = None*, ***otherkwargs*) → dict[str, tuple | ndarray]

Color argument processing for pyrolite plots, returning a standardised output.

Parameters

- **c** (*str | list | tuple | numpy.ndarray*) – Color arguments as typically passed to *matplotlib.pyplot.scatter*.
- **color** (*str | list | tuple | numpy.ndarray*) – Color arguments as typically passed to *matplotlib.pyplot.plot*
- **cmap** (str | *~matplotlib.cm.ScalarMappable*) – Colormap for mapping unknown color values.
- **alpha** (*float*) – Alpha to modulate color opacity.
- **norm** (*~matplotlib.colors.Normalize*) – Normalization for the colormap.
- **cmap_under** (*str | tuple*) – Color for values below the lower threshold for the cmap.
- **color_converter** – Function to use to convert colors (from strings, hex, tuples etc).
- **color_mappings** (*dict*) – Dictionary containing category-color mappings for individual color variables, with the default color mapping having the key 'color'. For use where categorical values are specified for a color variable.
- **size** (*int*) – Size of the data array along the first axis.

Returns
C – Color returned in standardised RGBA format.

Return type
tuple | numpy.ndarray

Notes

As formulated here, the addition of unused styling parameters may cause some properties (associated with 'c') to be set to None - and hence revert to defaults. This might be mitigated if the context could be checked - e.g. via checking keyword argument membership of *~pyrolite.util.plot.style.scatterkwargs* etc.

6.1.2 pyrolite.geochem

Submodule for working with geochemical data.

pyrolite.geochem.pyrochem (Pandas Interface)

class pyrolite.geochem.**pyrochem**(*obj: Series | DataFrame*)

property list_elements: list[str]
Get the subset of columns which are element names.

Return type
list

Notes

The list will have the same ordering as the source DataFrame.

property list_isotope_ratios: list[str]
Get the subset of columns which are isotope ratios.

Return type
list

Notes

The list will have the same ordering as the source DataFrame.

property list_REE: list[str]

Get the subset of columns which are Rare Earth Element names.

Return type
list

Notes

The returned list will reorder REE based on atomic number.

property list_REY: list[str]

Get the subset of columns which are Rare Earth Element names.

Return type
list

Notes

The returned list will reorder REE based on atomic number.

property list_oxides: list[str]

Get the subset of columns which are oxide names.

Return type
list

Notes

The list will have the same ordering as the source DataFrame/Series.

property list_compositional: list[str]

property elements: DataFrame

Get an elemental subset of a DataFrame.

Return type
pandas.DataFrame | pandas.Series

property REE: DataFrame

Get a Rare Earth Element subset of a DataFrame.

Return type
pandas.DataFrame | pd.Series

property REY: DataFrame

Get a Rare Earth Element + Yttrium subset of a DataFrame.

Return type
pandas.DataFrame | pandas.Series

property oxides: DataFrame

Get an oxide subset of a DataFrame.

Return type
pandas.DataFrame | pandas.Series

property isotope_ratios: DataFrame

Get an isotope ratio subset of a DataFrame.

Return type
pandas.DataFrame | pandas.Series

property compositional: DataFrame

Get an oxide & elemental subset of a DataFrame.

Return type
pandas.DataFrame | pandas.Series

Notes

This wil not include isotope ratios.

parse_chem(*abbrv: list[str] | None = None*, *split_on: str = '[\\s_]+'*) → DataFrame | Series

Convert column names to pyrolite-recognised elemental, oxide and isotope ratio column names where valid names are found.

check_multiple_cation_inclusion(*exclude: list[str] | None = None*) → set

Returns cations which are present in both oxide and elemental form.

Parameters
exclude (*list*) – List of components to exclude from the duplication check.

Returns
Set of elements for which multiple components exist in the dataframe.

Return type
set

to_molecular(*renorm: bool = True*) → DataFrame | Series

Converts mass quantities to molar quantities.

Parameters
renorm (*bool*) – Whether to renormalise the dataframe after converting to relative moles.

Notes

Does not convert units (i.e. mass% –> mol%; mass-ppm –> mol-ppm).

Returns
Transformed dataframe.

Return type
pandas.DataFrame | pandas.Series

to_weight(*renorm: bool = True*) → DataFrame | Series

Converts molar quantities to mass quantities.

Parameters
renorm (*bool*) – Whether to renormalise the dataframe after converting to relative moles.

Notes

Does not convert units (i.e. mol% –> mass%; mol-ppm –> mass-ppm).

Returns
Transformed dataframe.

Return type
pandas.DataFrame | pandas.Series

devolatilise(*exclude: list[str] | None = None*, *renorm: bool = True*) → DataFrame | Series

Recalculates components after exclusion of volatile phases (e.g. H2O, CO2).

Parameters

- **exclude** (*list*) – Components to exclude from the dataset.
- **renorm** (*bool*) – Whether to renormalise the dataframe after devolatilisation.

Returns
Transformed dataframe.

Return type
pandas.DataFrame | pandas.Series

elemental_sum(*component: str | None = None*, *to: str | None = None*, *total_suffix: str = 'T'*, *logdata: bool = False*, *molecular: bool = False*) → Series | float

Sums abundance for a cation to a single series, starting from a dataframe containing multiple componnents with a single set of units.

Parameters

- **component** (*str*) – Component indicating which element to aggregate.
- **to** (*str*) – Component to cast the output as.
- **logdata** (*bool*) – Whether data has been log transformed.
- **molecular** (*bool*) – Whether to perform a sum of molecular data.

Returns
Series with cation aggregated.

Return type
pandas.Series | float

aggregate_element(*to: str | Element | Formula | dict*, *total_suffix: str = 'T'*, *logdata: bool = False*, *renorm: bool = False*, *molecular: bool = False*) → Series | float

Aggregates cation information from oxide and elemental components to either a single species or a designated mixture of species.

Parameters

- **to** (*str | periodictable.core.Element | periodictable.formulas.Formula | dict*) – Component(s) to convert to. If one component is specified, the element will be converted to the target species.

  If more than one component is specified with proportions in a dictionary (e.g. *{'FeO': 0.9, 'Fe2O3': 0.1}*), the components will be split as a fraction of the elemental sum.
- **renorm** (*bool*) – Whether to renormalise the dataframe after recalculation.
- **total_suffix** (*str*) – Suffix of 'total' variables. E.g. 'T' for FeOT, Fe2O3T.
- **logdata** (*bool*) – Whether the data has been log transformed.
- **molecular** (*bool*) – Whether to perform a sum of molecular data.

Notes

This won't convert units, so need to start from single set of units.

Returns
Series with cation aggregated.

Return type
pandas.Series

recalculate_Fe(*tol: str | Element | Formula | dict = 'FeOT'*, *renorm: bool = False*, *total_suffix: str = 'T'*, *logdata: bool = False*, *molecular: bool = False*) → DataFrame | Series

Recalculates abundances of iron, and normalises a dataframe to contain either a single species, or multiple species in certain proportions.

Parameters

- **to** (*str | periodictable.core.Element | periodictable.formulas.Formula | dict*) – Component(s) to convert to.

  If one component is specified, all iron will be converted to the target species.

  If more than one component is specified with proportions in a dictionary (e.g. *{'FeO': 0.9, 'Fe2O3': 0.1}*), the components will be split as a fraction of Fe.
- **renorm** (*bool*) – Whether to renormalise the dataframe after recalculation.
- **total_suffix** (*str*) – Suffix of 'total' variables. E.g. 'T' for FeOT, Fe2O3T.
- **logdata** (*bool*) – Whether the data has been log transformed.
- **molecular** (*bool*) – Flag that data is in molecular units, rather than weight units.

Returns
Transformed dataframe.

Return type
pandas.DataFrame

get_ratio(*ratio: str*, *alias: str | None = None*, *norm_to: str | Composition | None = None*, *molecular: bool = False*) → Series | float

Add a ratio of components A and B, given in the form of string 'A/B'. Returned series be assigned an alias name.

Parameters

- **ratio** (*str*) – String decription of ratio in the form A/B[_n].
- **alias** (*str*) – Alternate name for ratio to be used as column name.
- **norm_to** (*str | pyrolite.geochem.norm.Composition*) – Reference composition to normalise to.
- **molecular** (*bool*) – Flag that data is in molecular units, rather than weight units.

Returns
Ratio series.

Return type
pandas.Series | float

See also

add_MgNo()

add_ratio(*ratio: str*, *alias: str | None = None*, *norm_to=None*, *molecular=False*) → DataFrame | Series

Add a ratio of components A and B, given in the form of string 'A/B'. Returned series be assigned an alias name.

Parameters

- **ratio** (*str*) – String decription of ratio in the form A/B[_n].
- **alias** (*str*) – Alternate name for ratio to be used as column name.
- **norm_to** (*str | pyrolite.geochem.norm.Composition*) – Reference composition to normalise to.
- **molecular** (*bool*) – Flag that data is in molecular units, rather than weight units.

Returns
Dataframe with ratio appended.

Return type
pandas.DataFrame

See also

add_MgNo()

add_MgNo(*molecular: bool = False*, *use_total_approx: bool = False*, *approx_Fe203_frac: float = 0.1*, *name: str = 'Mg#'*) → DataFrame | Series

Append the magnesium number to a dataframe.

Parameters

- **molecular** (bool, *False*) – Whether the input data is molecular.
- **use_total_approx** (bool, *False*) – Whether to use an approximate calculation using total iron rather than just FeO.
- **approx_Fe203_frac** (*float*) – Fraction of iron which is oxidised, used in approximation mentioned above.
- **name** (*str*) – Name to use for the Mg Number column.

Returns
Dataframe with ratio appended.

Return type
pandas.DataFrame

See also

add_ratio()

lambda_lnREE(*norm_to: str | Composition | None = 'ChondriteREE_ON'*, *exclude: list[str] | None = None*, *params: list | str | None = None*, *degree: int = 4*, *scale: str = 'ppm'*, *sigmas: ndarray[tuple[int], dtype[floating]] | None = None*, ***kwargs*) → DataFrame | Series

Calculates orthogonal polynomial coefficients (lambdas) for a given set of REE data, normalised to a specific composition [#pyrolite.geochem.lambda_lnREE_1]_. Lambda factors are given for the radii vs. ln(REE/NORM) polynomial combination.

Parameters

- **norm_to** (*str | pyrolite.geochem.norm.Composition | numpy.ndarray*) – Which reservoir to normalise REE data to (defaults to *"ChondriteREE_ON"*).
- **exclude** (*list*) – Which REE elements to exclude from the *fit*. May wish to include Ce for minerals in which Ce anomalies are common.
- **params** (*list | str*) – Pre-computed parameters for the orthogonal polynomials (a list of tuples). Optionally specified, otherwise defaults the parameterisation as in O'Neill (2016). If a string is supplied, *"O'Neill (2016)"* or similar will give the original defaults, while *"full"* will use all of the REE (including Eu) as a basis for the orthogonal polynomials.
- **degree** (*int, 4*) – Maximum degree polynomial fit component to include.
- **scale** (*str*) – Current units for the REE data, used to scale the reference dataset.
- **sigmas** (*float | numpy.ndarray | pandas.Series*) – Value or 1D array of fractional REE uncertaintes (i.e. σ_{REE}/REE).

References

See also

get_ionic_radii() :func:~pyrolite.util.lambdas.calc_lambdas` `orthogonal_polynomial_constants()` `REE_radii_plot()`

convert_chemistry(*to: list[str | dict] | None = None*, *logdata: bool = False*, *renorm: bool = False*, *molecular: bool = False*) → DataFrame | Series

Attempts to convert a dataframe with one set of components to another.

Parameters

- **to** (*list*) – Set of columns to try to extract from the dataframe.

  Can also include a dictionary for iron speciation. See `pyrolite.geochem.recalculate_Fe()`.
- **logdata** (*bool*) – Whether chemical data has been log transformed. Necessary for aggregation functions.
- **renorm** (*bool*) – Whether to renormalise the data after transformation.
- **molecular** (*bool*) – Flag that data is in molecular units, rather than weight units.

Returns
Dataframe with converted chemistry.

Return type
pandas.DataFrame

Todo

- Check for conflicts between oxides and elements

- Aggregator for ratios
- Implement generalised redox transformation.
- Add check for dicitonary components (e.g. Fe) in tests

normalize_to(*reference: str | Composition | ndarray[tuple[int], dtype[floating]] | None = None*, *units: str | None = None*, *convert_first: bool = False*) → DataFrame | Series

Normalise a dataframe to a given reference composition.

Parameters

- **reference** (*str | pyrolite.geochem.norm.Composition | numpy.ndarray*) – Reference composition to normalise to.
- **units** (*str*) – Units of the input dataframe, to convert the reference composition.
- **convert_first** (*bool*) – Whether to first convert the referenece compostion before normalisation. This is useful where elements are presented as different components (e.g. Ti, TiO2).

Returns
Dataframe with normalised chemistry.

Return type
pandas.DataFrame

Notes

This assumes that dataframes have a single set of units.

denormalize_from(*reference: str | Composition | ndarray[tuple[int], dtype[floating]]*, *units: str | None = None*) → DataFrame | Series

De-normalise a dataframe from a given reference composition.

Parameters

- **reference** (str | *~pyrolite.geochem.norm.Composition* | numpy.ndarray) – Reference composition which the composition is normalised to.
- **units** (*str*) – Units of the input dataframe, to convert the reference composition.

Returns
Dataframe with normalised chemistry.

Return type
pandas.DataFrame

Notes

This assumes that dataframes have a single set of units.

scale(*in_unit: str*, *target_unit: str = 'ppm'*) → DataFrame | Series

Scale a dataframe from one set of units to another.

Parameters

- **in_unit** (*str*) – Units to be converted from
- **target_unit** (str, *"ppm"*) – Units to scale to.

Returns
Dataframe with new scale.

Return type
pandas.DataFrame

pyrolite.geochem.ind

Collections and indexes of elements and oxides, and functions for obtaining these and relevant properties (e.g. radii).

Todo

- Incompatibility indexes for spider plot ordering.

pyrolite.geochem.ind.**common_elements**(*cutoff: int = 92*, *output: str = 'string'*, *order: Callable | None = None*, *as_set: bool = False*) → list[str] | list[Element] | set[str] | set[Element]

Provides a list of elements up to a particular cutoff (by default including U).

Parameters

- **cutoff** (*int*) – Upper cutoff on atomic number for the output list. Defaults to stopping at uranium (92).
- **output** (*str*) – Whether to return output list as formulae ('formula') or strings (anthing else).
- **order** (*Callable*) – Sorting function for elements.
- **as_set** (bool, *False*) – Whether to return a set (*True*) or list (*False*).

Returns
List of elements.

Return type
list | set

Notes

Formulae cannot be used as members of a set, and hence sets returned will instead consist only of strings.

Todo

- Implement ordering for e.g. incompatibility.

pyrolite.geochem.ind.**REE**(*output: str = 'string'*, *dropPm: bool = True*) → list[str] | list[Element]

Provides a list of Rare Earth Elements.

Parameters

- **output** (*str*) – Whether to return output list as formulae ('formula') or strings (anthing else).
- **dropPm** (*bool*) – Whether to exclude the (almost) non-existent element Promethium from the REE list.

Returns
List of REE.

Return type
list | set

pyrolite.geochem.ind.**REY**(*output: str = 'string'*, *dropPm: bool = True*) → list[str] | list[Element]

Provides a list of Rare Earth Elements, with the additiona of Yttrium.

Parameters

- **output** (*str*) – Whether to return output list as formulae ('formula') or strings (anthing else).
- **dropPm** (*bool*) – Whether to exclude the (almost) non-existent element Promethium from the REE list.

Returns
List of REE+Y.

Return type
list | set

Notes

This currently modifies the hardcoded list of *REE*, but could be adapated for different element ordering.

pyrolite.geochem.ind.**common_oxides**(*elements: list | None = None*, *output='string'*, *addition: list | None = None*, *exclude=None*, *as_set=False*) → list[str] | list[Formula] | set[str] | set[Formula]

Creates a list of oxides based on a list of elements.

Parameters

- **elements** (*list*) – List of elements to obtain oxide forms for.
- **output** (*str*) – Whether to return output list as formulae ('formula') or strings (anthing else).
- **addition** (*list*) – Additional components to append to the list.
- **exclude** (*list*) – Elements to not produce oxide forms for (e.g. oxygen, noble gases).
- **as_set** (*bool*) – Whether to return a set (*True*) or list (*False*).

Returns
List of oxides.

Return type
list | set

Notes

Formulae cannot be used as members of a set, and hence sets returned will instead consist only of strings.

Todo

- Element verification
- Conditional additional components on the presence of others (e.g. Fe - FeOT)

pyrolite.geochem.ind.**simple_oxides**(*cation: str | Element*, *output='string'*) → list[str] | list[Formula]

Creates a list of oxides for a cationic element (oxide of ions with c=1+ and above).

Parameters

- **cation** (*str | periodictable.core.Element*) – Cation to obtain oxide forms for.
- **output** (*str*) – Whether to return output list as formulae ('formula') or strings (anthing else).

Returns
List of oxides.

Return type
list | set

pyrolite.geochem.ind.**get_cations**(*component: str*, *exclude: list[str] | None = None*, *total_suffix: str = 'T'*) → list[Element]

Returns the principal cations in an oxide component.

Parameters

- **component** (str | *periodictable.formulas.Formula*) – Component to obtain cations for.
- **exclude** (*list*) – Components to exclude, i.e. anions (e.g. O, Cl, F).

Returns
List of cations.

Return type
list

Todo

- Consider implementing *periodictable.core.Element* return.

pyrolite.geochem.ind.**get_isotopes**(*ratio_text: str*) → list[str]

Regex for isotope ratios.

Parameters
ratio_text (*str*) – Text to extract isotope ratio components from.

Returns
Isotope ration numerator and denominator.

Return type
list

pyrolite.geochem.ind.**by_incompatibility**(*els: list[str]*, *reverse: bool = False*) → list[str]

Order a list of elements by their relative 'incompatibility' given by a proxy of the relative abundances in Bulk Continental Crust over a Primitive Mantle Composition.

Parameters

- **els** (*list*) – List of element names to be reodered.
- **reverse** (*bool*) – Whether to reverse the ordering.

Returns
Reordered list of elements.

Return type
list

Notes

Some elements are missing from this list, as as such will be omitted.

pyrolite.geochem.ind.**by_number**(*els: list[str]*, *reverse: bool = False*) → list[str]

Order a list of elements by their atomic number.

Parameters

- **els** (*list*) – List of element names to be reodered.
- **reverse** (*bool*) – Whether to reverse the ordering.

Returns

Reordered list of elements.

Return type

list

pyrolite.geochem.ind.**get_ionic_radii**(*element: str | list[str]*, *charge: int | None = None*, *coordination: int | None = None*, *variant: list[str] | None = None*, *source: str = 'shannon'*, *pauling: bool = True*, ***kwargs*) → ndarray | Series | float

Function to obtain ionic radii for a given ion and coordination [#pyrolite.geochem.ind.get_ionic_radii_1]_ [#pyrolite.geochem.ind.get_ionic_radii_2]_.

Parameters

- **element** (*str | list*) – Element to obtain a radii for. If a list is passed, the function will be applied over each of the items.
- **charge** (*int*) – Charge of the ion to obtain a radii for. If unspecified will use the default charge from `pyrolite.mineral.ions`.
- **coordination** (*int*) – Coordination of the ion to obtain a radii for.
- **variant** (*list*) – List of strings specifying particular variants (here 'squareplanar' or 'pyramidal', 'highspin' or 'lowspin').
- **source** (*str*) – Name of the data source for ionic radii ('shannon' [#pyrolite.geochem.ind.get_ionic_radii_1]_ or 'whittaker' [#pyrolite.geochem.ind.get_ionic_radii_2]_).
- **pauling** (*bool*) – Whether to use the radii consistent with Pauling (1960) [#pyrolite.geochem.ind.get_ionic_radii_3]_ from the Shannon (1976) radii dataset [#pyrolite.geochem.ind.get_ionic_radii_1]_.

Returns

Series with viable ion charge and coordination, with associated radii in angstroms. If the ion charge and coordiation are completely specified and found in the table, a single value will be returned instead.

Return type

pandas.Series | numpy.ndarray | float

Notes

Shannon published two sets of radii. The first ('Crystal Radii') were using Shannon's value for $r(O^{2-}_{VI})$ of 1.26 Å, while the second ('Ionic Radii') is consistent with the Pauling (1960) value of $r(O^{2-}_{VI})$ of 1.40 Å [#pyrolite.geochem.ind.get_ionic_radii_3]_.

References

Todo

- Implement interpolation for coordination +/- charge.

pyrolite.geochem.transform

Functions for converting, transforming and parameterizing geochemical data.

pyrolite.geochem.transform.**to_molecular**(*df: DataFrame*, *renorm: bool*) → DataFrame

pyrolite.geochem.transform.**to_molecular**(*df: Series*, *renorm: bool*) → Series

Converts mass quantities to molar quantities of the same order.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to transform.
- **renorm** (bool, *True*) – Whether to renormalise the dataframe after converting to relative moles.

Returns

Transformed dataframe.

Return type

pandas.DataFrame

Notes

Does not convert units (i.e. mass% –> mol%; mass-ppm –> mol-ppm).

pyrolite.geochem.transform.**to_weight**(*df: DataFrame*, *renorm: bool*) → DataFrame

pyrolite.geochem.transform.**to_weight**(*df: Series*, *renorm: bool*) → Series

Converts molar quantities to mass quantities of the same order.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to transform.
- **renorm** (bool, *True*) – Whether to renormalise the dataframe after converting to relative moles.

Returns

Transformed dataframe.

Return type

pandas.DataFrame

Notes

Does not convert units (i.e. mol% –> mass%; mol-ppm –> mass-ppm).

pyrolite.geochem.transform.**devolatilise**(*df: DataFrame*, *exclude: list[str]*, *renorm: bool*) → DataFrame

pyrolite.geochem.transform.**devolatilise**(*df: Series*, *exclude: list[str]*, *renorm: bool*) → Series

Recalculates components after exclusion of volatile phases (e.g. H2O, CO2).

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to devolatilise.

- **exclude** (*list*) – Components to exclude from the dataset.
- **renorm** (bool, *True*) – Whether to renormalise the dataframe after devolatilisation.

Returns
Transformed dataframe.

Return type
pandas.DataFrame

pyrolite.geochem.transform.**oxide_conversion**(*oxin: str | Formula*, *oxout: str | Formula*, *molecular: bool = False*) → Callable

Factory function to generate a function to convert oxide components between two elemental oxides, for use in redox recalculations.

Parameters

- **oxin** (*str | periodictable.formulas.Formula*) – Input component.
- **oxout** (*str | periodictable.formulas.Formula*) – Output component.
- **molecular** (bool, *False*) – Whether to apply the conversion for molecular data.

Returns

- *Function to convert a pandas.Series from one elment-oxide*
- *component to another.*

pyrolite.geochem.transform.**elemental_sum**(*df: DataFrame | Series*, *component: str*, *to: str | None = None*, *total_suffix: str = 'T'*, *logdata: bool = False*, *molecular: bool = False*) → Series

Sums abundance for a cation to a single series, starting from a dataframe containing multiple componnents with a single set of units.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame for which to aggregate cation data.
- **component** (*str*) – Component indicating which element to aggregate.
- **to** (*str*) – Component to cast the output as.
- **logdata** (bool, *False*) – Whether data has been log transformed.
- **molecular** (bool, *False*) – Whether to perform a sum of molecular data.

Returns
Series with cation aggregated.

Return type
pandas.Series

pyrolite.geochem.transform.**aggregate_element**(*df: DataFrame | Series*, *to: str | Element | Formula | dict*, *total_suffix: str = 'T'*, *logdata: bool = False*, *renorm: bool = False*, *molecular: bool = False*) → DataFrame

Aggregates cation information from oxide and elemental components to either a single species or a designated mixture of species.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame for which to aggregate cation data.

- **to** (str | ~*periodictable.core.Element* | periodictable.formulas.Formula | dict) – Component(s) to convert to. If one component is specified, the element will be converted to the target species.

  If more than one component is specified with proportions in a dictionary (e.g. *{'FeO': 0.9, 'Fe2O3': 0.1}*), the components will be split as a fraction of the elemental sum.
- **renorm** (bool, *True*) – Whether to renormalise the dataframe after recalculation.
- **total_suffix** (*str, 'T'*) – Suffix of 'total' variables. E.g. 'T' for FeOT, Fe2O3T.
- **logdata** (bool, *False*) – Whether the data has been log transformed.
- **molecular** (bool, *False*) – Whether to perform a sum of molecular data.

Notes

This won't convert units, so need to start from single set of units.

Returns
Dataframe with cation aggregated to the desired species.

Return type
pandas.DataFrame

pyrolite.geochem.transform.**get_ratio**(*df: DataFrame | Series*, *ratio: str*, *alias: str | None = None*, *norm_to: str | float | list | Composition | None = None*, *molecular: bool = False*) → Series

Get a ratio of components A and B, given in the form of string 'A/B'. Returned series be assigned an alias name.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to append ratio to.
- **ratio** (*str*) – String decription of ratio in the form A/B[_n].
- **alias** (*str*) – Alternate name for ratio to be used as column name.
- **norm_to** (str | *pyrolite.geochem.norm.Composition*, *None*) – Reference composition to normalise to.
- **molecular** (bool, *False*) – Flag that data is in molecular units, rather than weight units.

Returns
Series containing ratios.

Return type
pd.Series

Todo

- Use elemental sum from reference compositions
- Use sympy-like functionality to accept arbitrary input for calculation e.g. *"MgNo = Mg / (Mg + Fe)"*

See also

add_MgNo()

pyrolite.geochem.transform.**add_MgNo**(*df: DataFrame | Series*, *molecular: bool = False*, *use_total_approx: bool = False*, *approx_Fe203_frac: float = 0.1*, *name: str = 'Mg#'*) → DataFrame | Series

Append the magnesium number to a dataframe.

Parameters

- **df** (*pandas.DataFrame*) – Input dataframe.
- **molecular** (bool, *False*) – Whether the input data is molecular.
- **use_total_approx** (bool, *False*) – Whether to use an approximate calculation using total iron rather than just FeO.
- **approx_Fe203_frac** (*float*) – Fraction of iron which is oxidised, used in approximation mentioned above.
- **name** (*str*) – Name to use for the Mg Number column.

Returns

Dataframe with ratio appended.

Return type

pandas.DataFrame

See also

get_ratio()

pyrolite.geochem.transform.**lambda_lnREE**(*df: DataFrame | Series*, *norm_to: str = 'ChondriteREE_ON'*, *exclude: list[str] | None = None*, *params: list | None = None*, *degree: int = 4*, *scale: str = 'ppm'*, *allow_missing: bool = True*, *min_elements: int = 7*, *algorithm: str = 'ONeill'*, *sigmas: list[float] | ndarray | None = None*, ***kwargs*) → DataFrame | Series

Calculates orthogonal polynomial coefficients (lambdas) for a given set of REE data, normalised to a specific composition **[#pyrolite.geochem.transform.lambda_lnREE_1]_**. Lambda coefficeints are given for the polynomial regression of ln(REE/NORM) vs radii.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to calculate lambda coefficients for.
- **norm_to** (str | *~pyrolite.geochem.norm.Composition* | numpy.ndarray) – Which reservoir to normalise REE data to (defaults to *"ChondriteREE_ON"*).
- **exclude** (list, *["Pm", "Eu"]*) – Which REE elements to exclude from the *fit*. May wish to include Ce for minerals in which Ce anomalies are common.
- **params** (list | str, *None*) – Pre-computed parameters for the orthogonal polynomials (a list of tuples). Optionally specified, otherwise defaults the parameterisation as in O'Neill (2016). If a string is supplied, *"O'Neill (2016)"* or similar will give the original defaults, while *"full"* will use all of the REE (including Eu) as a basis for the orthogonal polynomials.
- **degree** (*int, 4*) – Maximum degree polynomial fit component to include.
- **scale** (*str*) – Current units for the REE data, used to scale the reference dataset.
- **allow_missing** (*True*) – Whether to calculate lambdas for rows which might be missing values.

- **min_elements** (*int*) – Minimum columns present to return lambda values.
- **algorithm** (*str*) – Algorithm to use for fitting the orthogonal polynomials.
- **sigmas** (*float* | *numpy.ndarray* | *pandas.Series*) – Value or 1D array of fractional REE uncertaintes (i.e. σ_{REE}/REE).

Todo

- Operate only on valid rows.
- Add residuals, Eu, Ce anomalies as options.

References

See also

get_ionic_radii(), *calc_lambdas()*, *orthogonal_polynomial_constants()*, REE_radii_plot()

pyrolite.geochem.transform.**convert_chemistry**(*input_df: Series | DataFrame*, *to: list[str | dict] | None = None*, *total_suffix: str = 'T'*, *renorm: bool = False*, *molecular: bool = False*, *logdata: bool = False*, ***kwargs*) → Series | DataFrame

Attempts to convert a dataframe with one set of components to another.

Parameters

- **input_df** (*pandas.Series* | *pandas.DataFrame*) – Dataframe to convert.
- **to** (*list*) – Set of columns to try to extract from the dataframe.

  Can also include a dictionary for iron speciation. See *aggregate_element*.
- **total_suffix** (*str, 'T'*) – Suffix of 'total' variables. E.g. 'T' for FeOT, Fe2O3T.
- **renorm** (bool, *False*) – Whether to renormalise the data after transformation.
- **molecular** (bool, *False*) – Flag that data is in molecular units, rather than weight units.
- **logdata** (bool, *False*) – Whether chemical data has been log transformed. Necessary for aggregation functions.

Returns
Dataframe with converted chemistry.

Return type
pandas.DataFrame

Todo

- Check for conflicts between oxides and elements
- Aggregator for ratios
- Implement generalised redox transformation.

- Add check for dicitonary components (e.g. Fe) in tests

pyrolite.geochem.norm

Reference compostitions and compositional normalisation.

pyrolite.geochem.norm.**set_DB**(*path: str | Path*)

Assign the database used for reference compositions, as per a given path.

class pyrolite.geochem.norm.**Composition**(*src: str | Path | DataFrame | Series | dict*, *name: str | None = None*, *reference: str | None = None*, *reservoir: str | None = None*, *source: str | None = None*, ***kwargs*)

set_units(*to: str = 'wt%'*)

Set the units of the dataframe.

Parameters
to (str, *"wt%"*)

describe(*verbose: bool = True*, ***kwargs*) → str

pyrolite.geochem.norm.**all_reference_compositions**() → dict[str, *Composition*]

Get a dictionary of all reference compositions indexed by name.

Return type
dict

pyrolite.geochem.norm.**get_reference_composition**(*name: str*) → *Composition*

Retrieve a particular composition from the reference database.

Parameters
name (*str*) – Name of the reference composition model.

Return type
pyrolite.geochem.norm.Composition

pyrolite.geochem.norm.**get_reference_files**(*directory: str | Path | None = None*, *formats: list[str] | None = None*) → list[Path]

Get a list of the reference composition files.

Parameters

- **directory** (str, *None*) – Location of reference data files.
- **formats** (list, *["csv"]*) – List of potential data formats to draw from. Currently only csv will work.

Return type
list

pyrolite.geochem.norm.**update_database**(*encoding: str = 'cp1252'*, ***kwargs*)

Update the reference composition database.

Notes

This will take all csv files from the geochem/refcomp pyrolite data folder and construct a document-based JSON database.

pyrolite.geochem.parse

Functions for parsing, formatting and validating chemical names and formulae.

pyrolite.geochem.parse.**is_isotoperatio**(*s: str*, *require_split: bool = False*, *split_on: str = '[\\s_]+'*) → bool

Check if text is plausibly an isotope ratio.

Parameters
s (*str*) – String to validate.

Return type
bool

Todo

- Validate the isotope masses vs natural isotopes

pyrolite.geochem.parse.**repr_isotope_ratio**(*isos: str | tuple[str]*) → str | None

Format an isotope ratio pair as a string.

Parameters
isotope_ratio (*tuple*) – Numerator, denominator pair.

Return type
str

Todo

Consider returning additional text outside of the match (e.g. 87Sr/86Sri should include the 'i').

pyrolite.geochem.parse.**ischem**(*s: str | list[str]*) → bool | list[bool]

Checks if a string corresponds to chemical component (compositional). Here simply checking whether it is a common element or oxide.

Parameters
s (*str*) – String to validate.

Return type
bool

Todo

- Implement checking for other compounds, e.g. carbonates.

pyrolite.geochem.parse.**tochem**(*strings: str | list[str] | Index*, *abbrv: list[str] | None = None*, *split_on: str = '[\\s_]+'*) → list[str] | str

Converts a list of strings containing come chemical compounds to appropriate case.

Parameters

- **strings** (*list*) – Strings to convert to 'chemical case'.
- **abbr** (list, *["ID", "IGSN"]*) – Abbreivated phrases to ignore in capitalisation.

- **split_on** (*str, "[s_]+"*) – Regex for character or phrases to split the strings on.

Return type
list | str

pyrolite.geochem.parse.**check_multiple_cation_inclusion**(*df: DataFrame | Series*, *exclude: list[str] | None = None*) → set[Element]

Returns cations which are present in both oxide and elemental form.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to check duplication within.
- **exclude** (list, *["LOI", "FeOT", "Fe2O3T"]*) – List of components to exclude from the duplication check.

Returns
Set of elements for which multiple components exist in the dataframe.

Return type
set

Todo

- Options for output (string/formula).

pyrolite.geochem.magma

Submodule for calculating and modelling melt chemistry. Includes common functions for predicting and accounting for melt evolution.

pyrolite.geochem.magma.**FeAt8MgO**(*FeOT: float | ndarray*, *MgO: float | ndarray*) → float | ndarray

To account for differences in the slopes and curvature of liquid lines of descent as a function of parental magma composition **[#pyrolite.geochem.magma.FeAt8MgO_1]_ [#pyrolite.geochem.magma.FeAt8MgO_2]_** (after **[#pyrolite.geochem.magma.FeAt8MgO_3]_**).

Parameters

- **FeOT** (*float*) – Iron oxide content.
- **MgO** (*float*) – Magnesium oxide content.

References

pyrolite.geochem.magma.**NaAt8MgO**(*Na2O: float | ndarray*, *MgO: float | ndarray*) → float | ndarray

To account for differences in the slopes and curvature of liquid lines of descent as a function of parental magma composition **[#pyrolite.geochem.magma.NaAt8MgO_1]_ [#pyrolite.geochem.magma.NaAt8MgO_2]_** (after **[#pyrolite.geochem.magma.NaAt8MgO_3]_**).

Parameters

- **Na2O** (*float*) – Iron oxide content.
- **MgO** (*float*) – Magnesium oxide content.

References

pyrolite.geochem.magma.**SCSS**(*df: DataFrame*, *T: float | ndarray*, *P: float | ndarray*, *kelvin: bool = False*, *grid: str | None = None*, *outunit: str = 'wt%'*) → tuple[ndarray, ndarray]

Obtain the sulfur content at sulfate and sulfide saturation **[#pyrolite.geochem.magma.SCSS_1]_ [#pyrolite.geochem.magma.SCSS_2]_**.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe of compositions.
- **T** (*float | numpy.ndarray*) – Temperature
- **P** (*float | numpy.ndarray*) – Pressure (kbar)
- **kelvin** (*bool*) – Whether temperature values are in kelvin (*True*) or celsuis (*False*)
- **grid** (*str |none*) – Whether to consider temperature and pressure as a geotherm (*geotherm*), or independently (as a grid, *grid*).

Returns

sulfate, sulfide – Arrays of mass fraction sulfate and sulfide abundances at saturation.

Return type

numpy.ndarray, numpy.ndarray

Notes

For anhydrite-saturated systems, the sulfur content at sulfate saturation is given by the following:

$$ln(X_S) = 10.07 - 1.151 \cdot (10^4/T_K) + 0.104 \cdot P_{kbar} \tag{6.1}$$
$$- 7.1 \cdot X_{SiO_2} - 14.02 \cdot X_{MgO} - 14.164 \cdot X_{Al_2O_3} \tag{6.2}$$
$$\tag{6.3}$$

For sulfide-liquid saturated systems, the sulfur content at sulfide saturation is given by the following:

$$ln(X_S) = -1.76 - 0.474 \cdot (10^4/T_K) + 0.021 \cdot P_{kbar} \tag{6.4}$$
$$+ 5.559 \cdot X_{FeO} + 2.565 \cdot X_{TiO_2} + 2.709 \cdot X_{CaO} \tag{6.5}$$
$$- 3.192 \cdot X_{SiO_2} - 3.049 \cdot X_{H_2O} \tag{6.6}$$
$$\tag{6.7}$$

References

Todo

- Produce an updated version based on log-regressions?
- Add updates from Smythe et al. (2017)?

pyrolite.geochem.alteration

Functions for calcuating indexes of chemical alteration.

pyrolite.geochem.alteration.**CIA**(*df: DataFrame*) → Series

Chemical Index of Alteration (molecular) **[#pyrolite.geochem.alteration.CIA_1]_**

Parameters

df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

References

pyrolite.geochem.alteration.**CIW**(*df: DataFrame*) → Series

Chemical Index of Weathering (molecular) **[#pyrolite.geochem.alteration.CIW_1]_**

Parameters
df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

References

pyrolite.geochem.alteration.**PIA**(*df: DataFrame*) → Series

Plagioclase Index of Alteration (molecular) **[#pyrolite.geochem.alteration.PIA_1]_**

Parameters
df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

References

pyrolite.geochem.alteration.**SAR**(*df: DataFrame*) → Series

Silica-Alumina Ratio (molecular)

Parameters
df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

pyrolite.geochem.alteration.**SiTiIndex**(*df: DataFrame*) → Series

Silica-Titania Index (molecular) **[#pyrolite.geochem.alteration.SiTiIndex_1]_**

Parameters
df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

References

pyrolite.geochem.alteration.**WIP**(*df: DataFrame*) → Series

Weathering Index of Parker (molecular) [#pyrolite.geochem.alteration.WIP_1]_

Parameters
df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

References

pyrolite.geochem.alteration.**IshikawaAltIndex**(*df: DataFrame*) → Series

Alteration Index of Ishikawa (wt%) [#pyrolite.geochem.alteration.IshikawaAltIndex_1]_

Parameters
df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

References

pyrolite.geochem.alteration.**CCPI**(*df: DataFrame*) → Series

Chlorite-carbonate-pyrite index of Large et al. (wt%) [#pyrolite.geochem.alteration.CCPI_1]_.

Parameters
df (*pandas.DataFrame*) – DataFrame to calculate index from.

Returns
Alteration index series.

Return type
pandas.Series

Notes

Here FeO is taken as total iron (FeO+Fe2O3).

References

pyrolite.geochem.ions

pyrolite.geochem.ions.**set_default_ionic_charges**(*charges: dict | None = None*)

Set the default ionic charges for each element.

Parameters
charges (*dict*) – Dictionary of elements : charges.

pyrolite.geochem.isotope

Submodule for calculation and transformation of mass spectrometry data (particularly for ion-counting and isotope ratio data). Currently in the early stages of development.

Todo

- Dodson ratiosDodson M. H. (1978) A linear method for second-degree interpolation in cyclical data collection. Journal of Physics E: Scientific Instruments 11, 296. doi: 10.1088/0022-3735/11/4/004[0]
- Add noise-corrected quasi-unbiased ratios of Coath et al. (2012)[?]
- Deadtime methods[???]
- Synthetic data, file format parsing
- Capability for dealing with pyrolite.geochem.isotopeerence materials

  This would be handy for *pyrolite.geochem* in general, and could be implemented in a similar way to how *pyrolite.geochem.norm* handles pyrolite.geochem.isotopeerence compositions.
- Stable isotope calculations
- Simple radiogenic isotope system calculations and plots
- U-Pb, U-series data reduction and uncertainty propagation

References

6.1.3 pyrolite.comp

Submodule for working with compositional data.

pyrolite.comp.pyrocomp (Pandas Interface)

class pyrolite.comp.**pyrocomp**(*obj: DataFrame | Series*)

renormalise(*components: list[str] | None = None*, *scale: float = 100.0*) → DataFrame | Series

Renormalises compositional data to ensure closure.

Parameters

- **components** (*list*) – Option subcompositon to renormalise to 100. Useful for the use case where compostional data and non-compositional data are stored in the same dataframe.
- **scale** (*float*) – Closure parameter. Typically either 100 or 1.

Returns

Renormalized dataframe.

Return type

pandas.DataFrame

[0] Dodson M. H. (1978) A linear method for second-degree interpolation in cyclical data collection. Journal of Physics E: Scientific Instruments 11, 296. doi: 10.1088/0022-3735/11/4/004

Notes

This won't modify the dataframe in place, you'll need to assign it to something. If you specify components, those components will be summed to 100%, and others remain unchanged.

ALR(*components: list[str] | None = None*, *ind: int | str = -1*, *null_col: bool = False*, *label_mode: str = 'simple'*) → DataFrame | Series

Additive Log Ratio transformation.

Parameters

- **ind** (*int | str*) – Index or name of column used as denominator.
- **null_col** (*bool*) – Whether to keep the redundant column.

Returns
ALR-transformed array, of shape *(N, D-1)*.

Return type
pandas.DataFrame

inverse_ALR(*ind: int | str | None = None*, *null_col: bool = False*) → DataFrame | Series

Inverse Additive Log Ratio transformation.

Parameters

- **ind** (*int | str*) – Index or name of column used as denominator.
- **null_col** (*bool*) – Whether the array contains an extra redundant column (i.e. shape is *(N, D)*).

Returns
Inverse-ALR transformed array, of shape *(N, D)*.

Return type
pandas.DataFrame

CLR(*label_mode: str = 'simple'*) → DataFrame | Series

Centred Log Ratio transformation.

Parameters
label_mode (*str*) – Labelling mode for the output dataframe (*numeric*, *simple*, *LaTeX*). If you plan to use the outputs for automated visualisation and want to know which components contribute, use *simple* or *LaTeX*.

Returns
CLR-transformed array, of shape *(N, D)*.

Return type
pandas.DataFrame

inverse_CLR() → DataFrame | Series

Inverse Centred Log Ratio transformation.

Returns
Inverse-CLR transformed array, of shape *(N, D)*.

Return type
pandas.DataFrame

ILR(*label_mode: str = 'simple'*) → DataFrame | Series

Isometric Log Ratio transformation.

Parameters
label_mode (*str*) – Labelling mode for the output dataframe (*numeric*, *simple*, *LaTeX*). If you plan to use the outputs for automated visualisation and want to know which components contribute, use *simple* or *LaTeX*.

Returns
ILR-transformed array, of shape *(N, D-1)*

Return type
pandas.DataFrame

inverse_ILR(*X: ndarray | None = None*) → DataFrame | Series

Inverse Isometric Log Ratio transformation.

Parameters
X (*numpy.ndarray*) – Optional specification for an array from which to derive the orthonormal basis, with shape *(N, D)*.

Returns
Inverse-ILR transformed array, of shape *(N, D)*.

Return type
pandas.DataFrame

boxcox(*lmbda: number | None = None*, *lmbda_search_space: tuple[float, float] = (-1, 5)*, *search_steps: int = 100*, *return_lmbda: bool = False*) → DataFrame | Series

Box-Cox transformation.

Parameters

- **lmbda** (*numpy.number*) – Lambda value used to forward-transform values. If none, it will be calculated using the mean
- **lmbda_search_space** (*tuple*) – Range tuple (min, max).
- **search_steps** (*int*) – Steps for lambda search range.

Returns
Box-Cox transformed array.

Return type
pandas.DataFrame

inverse_boxcox(*lmbda: float | None = None*) → DataFrame | Series

Inverse Box-Cox transformation.

Parameters
lmbda (*float*) – Lambda value used to forward-transform values.

Returns
Inverse Box-Cox transformed array.

Return type
pandas.DataFrame

sphere() → DataFrame | Series

Spherical coordinate transformation for compositional data.

Returns
– Array of angles in radians ($(0, \pi/2]$)

Return type
pandas.DataFrame

inverse_sphere(*variables=None*) → DataFrame | Series

Inverse spherical coordinate transformation to revert back to compositional data in the simplex.

Parameters
variables (*list*) – List of names for the compositional data variables, optionally specified (for when they may not be stored in the dataframe attributes through the *pyrocomp()* functions).

Returns
df – Dataframe of original compositional (simplex) coordinates, normalised to 1.

Return type
pandas.DataFrame

logratiomean(*transform: Callable = <function CLR>*, *inverse_transform: Callable = <function inverse_CLR>*) → Series

Take a mean of log-ratios along the index of a dataframe.

Parameters

- **transform** (*Callable*) – Log transform to use.
- **inverse_transform** (*Callable*) – Inverse transform to use.

Returns
Mean values as a pandas series.

Return type
pandas.Series

Notes

Only makes sense for a dataframe.

invert_transform(***kwargs*) → Series | DataFrame

Try to inverse-transform a transformed dataframe.

pyrolite.comp.codata

pyrolite.comp.codata.**close**(*X: ndarray[tuple[int, int], dtype[number]]*, *sumf: Callable*) → ndarray[tuple[int, int], dtype[floating]]

pyrolite.comp.codata.**close**(*X: ndarray[tuple[int], dtype[number]]*, *sumf: Callable*) → ndarray[tuple[int], dtype[floating]]

Closure operator for compositional data.

Parameters

- **X** (*numpy.ndarray*) – Array to close.
- **sumf** (Callable, *numpy.sum*) – Sum function to use for closure.

Returns
Closed array.

Return type
numpy.ndarray

Notes

Checks for non-positive entries and replaces zeros with NaN values.

pyrolite.comp.codata.**renormalise**(*df: DataFrame*, *components: list[str] | None*, *scale: float*) → DataFrame

pyrolite.comp.codata.**renormalise**(*df: Series*, *components: list[str] | None*, *scale: float*) → Series

Renormalises compositional data to ensure closure.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to renomalise.
- **components** (*list*) – Option subcompositon to renormalise to 100. Useful for the use case where compostional data and non-compositional data are stored in the same dataframe.
- **scale** (*float*) – Closure parameter. Typically either 100 or 1.

Returns
Renormalized dataframe.

Return type
pandas.DataFrame

pyrolite.comp.codata.**ALR**(*X: ndarray[tuple[int, int], dtype[number]]*, *ind: int = -1*, *null_col=False*) → ndarray[tuple[int, int], dtype[floating]]

Additive Log Ratio transformation.

Parameters

- **X** (*numpy.ndarray*) – Array on which to perform the transformation, of shape *(N, D)*.
- **ind** (*int*) – Index of column used as denominator.
- **null_col** (*bool*) – Whether to keep the redundant column.

Returns
ALR-transformed array, of shape *(N, D-1)*.

Return type
numpy.ndarray

pyrolite.comp.codata.**inverse_ALR**(*Y: ndarray*, *ind=-1*, *null_col=False*) → ndarray[tuple[int, int], dtype[floating]]

Inverse Centred Log Ratio transformation.

Parameters

- **Y** (*numpy.ndarray*) – Array on which to perform the inverse transformation, of shape *(N, D-1)*.
- **ind** (*int*) – Index of column used as denominator.
- **null_col** (bool, *False*) – Whether the array contains an extra redundant column (i.e. shape is *(N, D)*).

Returns
Inverse-ALR transformed array, of shape *(N, D)*.

Return type
numpy.ndarray

pyrolite.comp.codata.**CLR**(*X: ndarray*) → ndarray[tuple[int, int], dtype[floating]]

Centred Log Ratio transformation.

Parameters
X (*numpy.ndarray*) – 2D array on which to perform the transformation, of shape *(N, D)*.

Returns
CLR-transformed array, of shape *(N, D)*.

Return type
numpy.ndarray

pyrolite.comp.codata.**inverse_CLR**(*Y: ndarray*) → ndarray[tuple[int, int], dtype[floating]]

Inverse Centred Log Ratio transformation.

Parameters
Y (*numpy.ndarray*) – Array on which to perform the inverse transformation, of shape *(N, D)*.

Returns
Inverse-CLR transformed array, of shape *(N, D)*.

Return type
numpy.ndarray

pyrolite.comp.codata.**ILR**(*X: ndarray*, *psi: ndarray | None = None*, ***kwargs*) → ndarray[tuple[int, int], dtype[floating]]

Isometric Log Ratio transformation.

Parameters

- **X** (*numpy.ndarray*) – Array on which to perform the transformation, of shape *(N, D)*.
- **psi** (*numpy.ndarray*) – Array or matrix representing the ILR basis; optionally specified.

Returns
ILR-transformed array, of shape *(N, D-1)*.

Return type
numpy.ndarray

pyrolite.comp.codata.**inverse_ILR**(*Y: ndarray*, *X: ndarray | None = None*, *psi: ndarray | None = None*, ***kwargs*) → ndarray[tuple[int, int], dtype[floating]]

Inverse Isometric Log Ratio transformation.

Parameters

- **Y** (*numpy.ndarray*) – Array on which to perform the inverse transformation, of shape *(N, D-1)*.
- **X** (numpy.ndarray, *None*) – Optional specification for an array from which to derive the orthonormal basis, with shape *(N, D)*.
- **psi** (*numpy.ndarray*) – Array or matrix representing the ILR basis; optionally specified.

Returns
Inverse-ILR transformed array, of shape *(N, D)*.

Return type
numpy.ndarray

pyrolite.comp.codata.**logratiomean**(*df: DataFrame*, *transform: Callable = <function CLR>*) → Series

Take a mean of log-ratios along the index of a dataframe.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe from which to compute a mean along the index.

- **transform** (*Callable*) – Log transform to use.
- **inverse_transform** (*Callable*) – Inverse of log transform.

Returns
Mean values as a pandas series.

Return type
pandas.Series

pyrolite.comp.codata.**get_ALR_labels**(*df: DataFrame*, *mode: str = 'simple'*, *ind: int = -1*, ***kwargs*) → list[str]

Get symbolic labels for ALR coordinates based on dataframe columns.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to generate ALR labels for.
- **mode** (*str*) – Mode of label to return (*LaTeX*, *simple*).

Returns
List of ALR coordinates corresponding to dataframe columns.

Return type
list

Notes

Some variable names are protected in `sympy` and if used can result in errors. If one of these column names is found, it will be replaced with a title-cased duplicated version of itself (e.g. 'S' will be replaced by 'Ss').

pyrolite.comp.codata.**get_CLR_labels**(*df: DataFrame*, *mode: str = 'simple'*, ***kwargs*) → list[str]

Get symbolic labels for CLR coordinates based on dataframe columns.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to generate CLR labels for.
- **mode** (*str*) – Mode of label to return (*LaTeX*, *simple*).

Returns
List of CLR coordinates corresponding to dataframe columns.

Return type
list

Notes

Some variable names are protected in `sympy` and if used can result in errors. If one of these column names is found, it will be replaced with a title-cased duplicated version of itself (e.g. 'S' will be replaced by 'Ss').

pyrolite.comp.codata.**get_ILR_labels**(*df: DataFrame*, *mode: str = 'latex'*, ***kwargs*) → list[str]

Get symbolic labels for ILR coordinates based on dataframe columns.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to generate ILR labels for.
- **mode** (*str*) – Mode of label to return (*LaTeX*, *simple*).

Returns
List of ILR coordinates corresponding to dataframe columns.

Return type
list

Notes

Some variable names are protected in `sympy` and if used can result in errors. If one of these column names is found, it will be replaced with a title-cased duplicated version of itself (e.g. 'S' will be replaced by 'Ss').

pyrolite.comp.codata.**boxcox**(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *lmbda: float | None = None*, *lmbda_search_space: tuple[float, float] = (-1, 5)*, *search_steps: int = 100*, *return_lmbda: bool = False*) → ndarray[tuple[int, int], dtype[floating]] | DataFrame | tuple[ndarray[tuple[int, int], dtype[floating]] | DataFrame, float]

Box-Cox transformation.

Parameters

- **X** (*numpy.ndarray*) – Array on which to perform the transformation.
- **lmbda** (*flaot*) – Lambda value used to forward-transform values. If none, it will be calculated using the mean.
- **lmbda_search_space** (*tuple*) – Range tuple (min, max).
- **search_steps** (*int*) – Steps for lambda search range.
- **return_lmbda** (*bool*) – Whether to also return the lambda value.

Returns
Box-Cox transformed array. If *return_lmbda* is true, tuple contains data and lambda value.

Return type
numpy.ndarray | pandas.DataFrame | tuple[np.ndarray | pandas.DataFrame,float]

pyrolite.comp.codata.**inverse_boxcox**(*Y: ndarray[tuple[int, int], dtype[floating]]*, *lmbda: float*) → ndarray[tuple[int, int], dtype[floating]]

Inverse Box-Cox transformation.

Parameters

- **Y** (*numpy.ndarray*) – Array on which to perform the transformation.
- **lmbda** (*float*) – Lambda value used to forward-transform values.

Returns
Inverse Box-Cox transformed array.

Return type
numpy.ndarray

pyrolite.comp.codata.**sphere**(*ys: ndarray[tuple[int, int], dtype[number]]*)

Spherical coordinate transformation for compositional data.

Parameters
ys (*numpy.ndarray*) – Compositional data to transform (shape (n, D)).

Returns
– Array of angles in radians ($(0, \pi/2]$)

Return type
numpy.ndarray

Notes

`numpy.arccos()` will return angles in the range $(0, \pi)$. This shouldn't be an issue for this function given that the input values are all positive.

`pyrolite.comp.codata.`**`inverse_sphere`**(*: ndarray[tuple[int, int], dtype[floating]]*)

Inverse spherical coordinate transformation to revert back to compositional data in the simplex.

Parameters
(*numpy.ndarray*) – Angular coordinates to revert.

Returns
ys – Compositional (simplex) coordinates, normalised to 1.

Return type
numpy.ndarray

`pyrolite.comp.codata.`**`compositional_cosine_distances`**(*arr: ndarray[tuple[int, int], dtype[floating]]*) → ndarray[tuple[int, int], dtype[floating]]

Calculate a distance matrix corresponding to the angles between a number of compositional vectors.

Parameters
arr (*numpy.ndarray*) – Array of n-dimensional compositions of shape (n_samples, n).

Returns
Array of angular distances of shape (n_samples, n_samples).

Return type
numpy.ndarray

`pyrolite.comp.codata.`**`get_transforms`**(*name: str | Callable*) → tuple[Callable, Callable]

Lookup a transform-inverse transform pair by name.

Parameters
name (*str*) – Name of of the transform pairs (e.g. 'CLR').

Returns
tfm, inv_tfm – Transform and inverse transform functions.

Return type
tuple[Callable,Callable]

pyrolite.comp.aggregate

`pyrolite.comp.aggregate.`**`get_full_column`**(*X: ndarray*) → int

Returns the index of the first array column which contains only finite numbers (i.e. no missing data, nan, inf).

Parameters
X (*numpy.ndarray*) – Array for which to find the first full column within.

Returns
Index of the first full column.

Return type
int

`pyrolite.comp.aggregate.`**`weights_from_array`**(*X: ndarray*) → ndarray[tuple[int], dtype[float32]]

Returns a set of equal weights with size equal to that of the first axis of an array.

Parameters
X (*numpy.ndarray*) – Array of compositions to produce weights for.

Returns
Array of weights.

Return type
numpy.ndarray

pyrolite.comp.aggregate.**nan_weighted_mean**(*X: ndarray[tuple[int, int], dtype[number]]*, *weights: ndarray[tuple[int], dtype[floating]] | None = None*) → ndarray[tuple[int, int]]

Returns a weighted mean of compositions, where weights are renormalised to account for missing data.

Parameters

- **X** (*numpy.ndarray*) – Array of compositions to take a weighted mean of.
- **weights** (*numpy.ndarray*) – Array of weights.

Returns
Array mean.

Return type
numpy.ndarray

pyrolite.comp.aggregate.**compositional_mean**(*df: DataFrame*, *weights: ndarray[tuple[int], dtype[floating]] | None = None*, ***kwargs*) → Series

Implements an aggregation using a compositional weighted mean.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe of compositions to aggregate.
- **weights** (*numpy.ndarray*) – Array of weights.

Returns
Mean values along index of dataframe.

Return type
pandas.Series

pyrolite.comp.aggregate.**nan_weighted_compositional_mean**(*X: ndarray[tuple[int] | tuple[int, int], dtype[number]]*, *weights: ndarray[tuple[int], dtype[floating]] | None = None*, *ind: int | None = None*, *renorm: bool = True*, ***kwargs*) → ndarray[tuple[int], dtype[floating]]

Implements an aggregation using a weighted mean, but accounts for nans. Requires at least one non-nan column for ALR mean.

When used for internal standardisation, there should be only a single common element - this would be used by default as the divisor here. When used for multiple-standardisation, the [specified] or first common element will be used.

Parameters

- **X** (*numpy.ndarray*) – Array of compositions to aggregate.
- **weights** (*numpy.ndarray*) – Array of weights.
- **ind** (*int*) – Index of the column to use as the ALR divisor.
- **renorm** (*bool*) – Whether to renormalise the output compositional mean to unity.

Returns
An array with the mean composition.

Return type
numpy.ndarray

pyrolite.comp.aggregate.**cross_ratios**(*df: DataFrame*) → ndarray[tuple[int, int, int], dtype[floating]]

Takes ratios of values across a dataframe, such that columns are denominators and the row indexes the numerators, to create a square array. Returns one array per record.

Parameters
df (*pandas.DataFrame*) – Dataframe of compositions to create ratios of.

Returns
A 3D array of ratios.

Return type
numpy.ndarray

pyrolite.comp.aggregate.**np_cross_ratios**(*X: ndarray*, *debug: bool = False*) → ndarray[tuple[int, int, int], dtype[floating]]

Takes ratios of values across an array, such that columns are denominators and the row indexes the numerators, to create a square array. Returns one array per record.

Parameters
X (*numpy.ndarray*) – Array of compositions to create ratios of.

Returns
A 3D array of ratios.

Return type
numpy.ndarray

pyrolite.comp.aggregate.**standardise_aggregate**(*df: DataFrame*, *int_std: str | None = None*, *fixed_record_idx: int = 0*, *renorm: bool = True*, ***kwargs*) → Series

Performs internal standardisation and aggregates dissimilar geochemical records. Note: this changes the closure parameter, and is generally intended to integrate major and trace element records.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe of compositions to aggregate of.
- **int_std** (*str*) – Name of the internal standard column.
- **fixed_record_idx** (*int*) – Numeric index of a specific record's for which to retain the internal standard value (e.g for standardising trace element data).
- **renorm** (*bool*) – Whether to renormalise to unity.

Returns
A series representing the internally standardised record.

Return type
pandas.Series

pyrolite.comp.impute

pyrolite.comp.impute.**EMCOMP**(*X: ~numpy.ndarray[tuple[int, int], ~numpy.dtype[~numpy.floating]]*, *threshold: ~numpy.ndarray[tuple[int], ~numpy.dtype[~numpy.floating]] | None = None*, *tol: float = 0.0001*, *convergence_metric: ~typing.Callable = <function <lambda>>*, *max_iter: int = 30*) → tuple[ndarray[tuple[int, int], dtype[floating]], float, int]

EMCOMP replaces rounded zeros in a compositional data set based on a set of thresholds. After Palarea-Albaladejo and Martín-Fernández (2008) [#ref_1]_.

Parameters

- **X** (*numpy.ndarray*) – Dataset with rounded zeros
- **threshold** (*numpy.ndarray*) – Array of threshold values for each component as a proprotion.
- **tol** (*float*) – Tolerance to check for convergence.
- **convergence_metric** (*Callable*) – Callable function to check for convergence. Here we use a compositional distance rather than a maximum absolute difference, with very similar performance. Function needs to accept two numpy.ndarray arguments and third tolerance argument.
- **max_iter** (*int*) – Maximum number of iterations before an error is thrown.

Returns

- **X_est** (*numpy.ndarray*) – Dataset with rounded zeros replaced.
- **prop_zeros** (*float*) – Proportion of zeros in the original data set.
- **n_iters** (*int*) – Number of iterations needed for convergence.

Notes

- At least one component without missing values is needed for the divisor. Rounded zeros/missing values are replaced by values below their respective detection limits.
- This routine is not completely numerically stable as written.

Todo

- Implement methods to deal with variable decection limits (i.e thresholds are array shape *(N, D)*)
- Conisder non-normal models for data distributions.
- Improve numerical stability to reduce the chance of *np.inf* appearing.

References

6.1.4 pyrolite.mineral

Note

This module is a work in progress and will be gradually updated.

A submodule for working with mineral data.

pyrolite.mineral.template

`class pyrolite.mineral.template.MineralTemplate(name, *components)`

set_structure(*components)

Set the structure of the mineral template.

Parameters

components – Argument list consisting of each of the structural components. Can consist of any mixture of Sites or argument tuples which can be passed to Site __init__.

copy()

class pyrolite.mineral.template.Mineral(*name=None*, *template=None*, *composition=None*, *endmembers=None*)

set_endmembers(*endmembers=None*)

Set the endmbmer components for a mineral.

add_endmember(*em*, *name=None*)

Add a single endmember to the database.

set_template(*template*, *name=None*)

Assign a mineral template to the mineral.

set_composition(*composition=None*)

Parse and assign a composition to the mineral.

Parameters

composition – Composition to assign to the mineral. Can be provided in any form which is digestable by parse_composition.

recalculate_cations(*composition=None*, *ideal_cations=None*, *ideal_oxygens=None*, *Fe_species=None*, *oxygen_constrained=False*)

Recalculate a composition to give an elemental ionic breakdown.

Parameters

- **composition** – Composition to recalculate. If not provided, will try to use the mineral composition as set.
- **ideal_cations** (*int*) – Ideal number of cations to use for formulae calcuations. Will only be used if oxygen is constrained (i.e. multiple Fe species present or oxygen_constrained=True).
- **ideal_oxygens** (*int*) – Ideal number of oxygens to use for formulae calcuations. Will only be used if oxygen is not constrained (i.e. single Fe species present and oxygen_constrained=False).
- **Fe_species** (*list*) – List of iron species for identifying redox-defined compositions.
- **oxygen_constrained** (*bool, False*) – Whether the oxygen is a closed or open system for the specific composition.

apfu()

Get the atoms per formula unit.

endmember_decompose(*det_lim=0.01*)

Decompose a mineral composition into endmember components.

Parameters

det_lim (*float*) – Detection limit for individual

Notes

Currently implmented using optimization based on mass fractions.

Todo

Implement site-based endmember decomposition, which will enable more checks and balances.

`calculate_occupancy`(*composition=None*, *error=1e-05*, *balances=None*)

Calculate the estimated site occupancy for a given composition. Ions will be assigned to sites according to affinities. Sites with equal affinities should recieve equal assignment.

Parameters

- **composition** – Composition to calculate site occupancy for.
- **error** (*float*) – Absolute error for floating point occupancy calculations.
- **balances** (*list*) – List of iterables containing ions to balance across multiple sites. Note that the partitioning will occur after non-balanced cations are assigned, and that ions are only balanced between sites which have defined affinities for all of the particular ions defined in the 'balance'.

`get_site_occupancy`()

Get the site occupancy for the mineral.

pyrolite.mineral.normative

pyrolite.mineral.normative.**unmix**(*comp: ndarray*, *parts: ndarray*, *order: int = 1*, *det_lim: float = 0.0001*) → ndarray

From a composition and endmember components, find a set of weights which best approximate the composition as a weighted sum of components.

Parameters

- **comp** (*numpy.ndarray*) – Array of compositions (shape *n_S, n_C*).
- **parts** (*numpy.ndarray*) – Array of endmembers (shape *n_E, n_C*).
- **order** (*int*) – Order of regularization, defaults to L1 for sparsity.
- **det_lim** (*float*) – Detection limit, below which minor components will be omitted for sparsity.

Returns

Array of endmember modal abundances (shape *n_S, n_E*)

Return type

numpy.ndarray

pyrolite.mineral.normative.**endmember_decompose**(*composition: DataFrame | Series | Formula | str*, *endmembers: str | list | dict | None = None*, *drop_zeros: bool = True*, *molecular: bool = True*, *order: int = 1*, *det_lim: float = 0.0001*) → DataFrame

Decompose a given mineral composition to given endmembers.

Parameters

- **composition** (*pandas.DataFrame | pandas.Series | periodictable.formulas.Formula | str*) – Composition to decompose into endmember components.
- **endmembers** (*str | list | dict*) – List of endmembers to use for the decomposition.

- **drop_zeros** (*bool*) – Whether to omit components with zero estimated abundance.
- **molecular** (*bool*) – Whether to *convert* the chemistry to molecular before calculating the decomposition.
- **order** (*int*) – Order of regularization passed to *unmix*, defaults to L1 for sparsity.
- **det_lim** (*float*) – Detection limit, below which minor components will be omitted for sparsity.

Return type
pandas.DataFrame

pyrolite.mineral.normative.**MiddlemostOxRatio**(*df: DataFrame*) → Series

Apply a TAS classification to a dataframe, and get estimated Fe2O3/FeO ratios from Middlemost (1989).

Parameters
df (*pandas.DataFrame*) – Dataframe to get Fe2O3/FeO ratios for, containing the required oxides to calculate TAS diagrams from (i.e. SiO2, Na2O, K2O).

Returns
ratios – Series of estimated Fe2O3/FeO ratios, based on TAS classification.

Return type
pandas.Series

References

Middlemost, Eric A. K. (1989). Iron Oxidation Ratios, Norms and the Classification of Volcanic Rocks. Chemical Geology 77, 1: 19-26. https://doi.org/10.1016/0009-2541(89)90011-9.

pyrolite.mineral.normative.**LeMaitreOxRatio**(*df: DataFrame*, *mode: str | None = None*) → Series

Parameters

- **df** (*pandas.DataFrame*) – Dataframe containing compositions to calibrate against.
- **mode** (*str*) – Mode for the correction - 'volcanic' or 'plutonic'.

Returns
Series with oxidation ratios.

Return type
pandas.Series

Notes

This is a $\mathrm{FeO}/(\mathrm{FeO} + \mathrm{Fe_2O_3})$ mass ratio, not a standard molar ratio $\mathrm{Fe^{2+}}/(\mathrm{Fe^{2+}} + \mathrm{Fe^{3+}})$ which is more straightfowardly used; data presented should be in mass units. For the calculation, SiO2, Na2O and K2O are expected to be present.

References

Le Maitre, R. W (1976). Some Problems of the Projection of Chemical Data into Mineralogical Classifications. Contributions to Mineralogy and Petrology 56, no. 2 (1 January 1976): 181-89. https://doi.org/10.1007/BF00399603.

pyrolite.mineral.normative.**Middlemost_Fe_correction**(*df: DataFrame*) → DataFrame

Parameters
df (*pandas.DataFrame*) – Dataframe containing compositions to calibrate against.

Returns
Series with two corrected iron components ($\mathrm{FeO}, \mathrm{Fe_2O_3}$).

Return type
pandas.DataFrame

References

Middlemost, Eric A. K. (1989). Iron Oxidation Ratios, Norms and the Classification of Volcanic Rocks. Chemical Geology 77, 1: 19-26. https://doi.org/10.1016/0009-2541(89)90011-9.

pyrolite.mineral.normative.**LeMaitre_Fe_correction**(*df: DataFrame*, *mode: str = 'volcanic'*) → DataFrame

Parameters

- **df** (*pandas.DataFrame*) – Dataframe containing compositions to correct iron for.
- **mode** (*str*) – Mode for the correction - 'volcanic' or 'plutonic'.

Returns
Series with two corrected iron components (FeO, Fe_2O_3).

Return type
pandas.DataFrame

References

Le Maitre, R. W (1976). Some Problems of the Projection of Chemical Data into Mineralogical Classifications. Contributions to Mineralogy and Petrology 56, no. 2 (1 January 1976): 181-89. https://doi.org/10.1007/BF00399603.

Middlemost, Eric A. K. (1989). Iron Oxidation Ratios, Norms and the Classification of Volcanic Rocks. Chemical Geology 77, 1: 19-26. https://doi.org/10.1016/0009-2541(89)90011-9.

pyrolite.mineral.normative.**CIPW_norm**(*df: DataFrame*, *Fe_correction: str | None = None*, *Fe_correction_mode: str | None = None*, *adjust_all_Fe: bool = False*, *return_adjusted_input: bool = False*, *return_free_components: bool = False*, *rounding: int = 3*) → DataFrame

Standardised calcuation of estimated mineralogy from bulk rock chemistry. Takes a dataframe of chemistry & creates a dataframe of estimated mineralogy. This is the CIPW norm of Verma et al. (2003). This version only uses major elements.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe containing compositions to transform.
- **Fe_correction** (*str*) – Iron correction to apply, if any. Will default to 'LeMaitre'.
- **Fe_correction_mode** (*str*) – Mode for the iron correction, where applicable.
- **adjust_all_Fe** (*bool*) – Where correcting iron compositions, whether to adjust all iron compositions, or only those where singular components are specified.
- **return_adjusted_input** (*bool*) – Whether to return the adjusted input chemistry with the output.
- **return_free_components** (*bool*) – Whether to return the free components in the output.
- **rounding** (*int*) – Rounding to be applied to input and output data.

Return type
pandas.DataFrame

References

Verma, Surendra P., Ignacio S. Torres-Alvarado, and Fernando Velasco-Tapia (2003). A Revised CIPW Norm. Swiss Bulletin of Mineralogy and Petrology 83, 2: 197-216. Verma, S. P., & Rivera-Gomez, M. A. (2013). Computer Programs for the Classification and Nomenclature of Igneous Rocks. Episodes, 36(2), 115-124.

Todo

- Note whether data needs to be normalised to 1 or 100?

Notes

The function expect oxide components to be in wt% and elemental data to be in ppm.

pyrolite.mineral.transform

pyrolite.mineral.transform.**formula_to_elemental**(*formula*, *weight=True*)

Convert a periodictable.formulas.Formula to elemental composition.

pyrolite.mineral.transform.**merge_formulae**(*formulas*)

Combine multiple formulae into one. Particularly useful for defining oxide mineral formulae.

Parameters
formulas (*iterable*) – Iterable of multiple formulae to merge into a single larger molecular formulae.

pyrolite.mineral.transform.**recalc_cations**(*df*, *ideal_cations=4*, *ideal_oxygens=6*, *Fe_species=None*, *oxygen_constrained=False*)

Recalculate a composition to a.p.f.u.

pyrolite.mineral.sites

class pyrolite.mineral.sites.**Site**(*name=None*, *coordination=0*, *affinities=None*, *mode='cation'*)

class pyrolite.mineral.sites.**MX**(*name='M'*, *coordination=8*, **args*, ***kwargs*)

Octahedrally coordinated M site.

class pyrolite.mineral.sites.**TX**(*name='T'*, *coordination=4*, *affinities=None*, **args*, *mode='cation'*, ***kwargs*)

Tetrahedrally coordinated T site.

class pyrolite.mineral.sites.**IX**(*name='I'*, *coordination=12*, **args*, ***kwargs*)

Dodecahedrally coordinated I site.

class pyrolite.mineral.sites.**VX**(*name='V'*, *coordination=0*, **args*, ***kwargs*)

Vacancy site.

class pyrolite.mineral.sites.**OX**(*name='O'*, *coordination=0*, *affinities=None*, **args*, ***kwargs*)

Oxygen site.

class pyrolite.mineral.sites.**AX**(*name='A'*, *coordination=0*, **args*, ***kwargs*)

Anion site.

pyrolite.mineral.lattice

Submodule for calcuating relative ion paritioning based on the lattice strain modelBrice, J.C., 1975. Some thermodynamic aspects of the growth of strained crystals. Journal of Crystal Growth 28, 249-253. doi: 10.1016/0022-0248(75)90241-9[0???].

Todo

- Bulk modulus and Youngs modulus approximations[???].

References

pyrolite.mineral.lattice.**strain_coefficient**(*ri: float*, *rx: float | ndarray*, *r0: float | None = None*, *E: float | None = None*, *T: float = 298.15*, *z: int | None = None*, ***kwargs*) → float

Calculate the lattice strain associated with an ionic substitution **[#pyrolite.mineral.lattice.strain_coefficient_1]_ [#pyrolite.mineral.lattice.strain_coefficient_2]_**.

Parameters

- **ri** (*float*) – Ionic radius to calculate strain relative to, in angstroms (Å).
- **rj** (*float | numpy.ndarray*) – Ionic radius to calculate strain for, in angstroms (Å).
- **r0** (*float*) – Fictive ideal ionic radii for the site. The value for *ri* will be used in its place if none is given, and a warning issued.
- **E** (*float*) – Young's modulus (stiffness) for the site, in pascals (Pa). Will be estimated using *youngs_modulus_approximation* if none is given.
- **T** (*float*) – Temperature, in Kelvin (K).
- **z** (*int*) – Optional specification of cationic valence, for calcuation of approximate Young's modulus using *youngs_modulus_approximation*, where the modulus is not specified.

Returns

The strain coefficent $e^{\frac{-\Delta G_{strain}}{RT}}$.

Return type

float

Notes

The lattice strain model relates changes in paritioning to differences in ionic radii for ions of a given cationic charge, and for a for a specific site (with Young's modulus E). This is calcuated using the work done to expand a spherical shell centred on the lattice site, which alters the ΔG for the formation of the mineral. This can be related to changes in partition coefficients using the following **[#pyrolite.mineral.lattice.strain_coefficient_2]_**:

$$D_{j^{n+}} = D_{A^{n+}} \cdot e^{\frac{-4\pi EN\left(\frac{r_0}{2}(r_j - r_0)^2 + \frac{1}{3}(r_j - r_0)^3\right)}{RT}}$$

Where $D_{A^{n+}}$ is the partition coefficient for the ideal ion A, and N is Avagadro's number (6.023e23 atoms/mol). This can also be calcuated relative to an 'ideal' fictive ion which has a maximum D where this data are available. This relationship arises via i) the integration to calcuate the strain energy mentioned above ($4\pi E(\frac{r_0}{2}(r_j - r_0)^2 + \frac{1}{3}(r_j - r_0)^3)$), and ii) the assumption that the changes in ΔG occur only to size differences, and the difference is

0

Brice, J.C., 1975. Some thermodynamic aspects of the growth of strained crystals. Journal of Crystal Growth 28, 249-253. doi: 10.1016/0022-0248(75)90241-9

additive. The 'segregation coefficient' K_j can be expressed relative to the non-doped equilibirum constant K_0 [#pyrolite.mineral.lattice.strain_coefficient_1]_:

$$K_j = e^{\frac{-\Delta G_0 - \Delta G_{strain}}{RT}} \tag{6.8}$$

$$= e^{\frac{-\Delta G_0}{RT}} \cdot e^{\frac{-\Delta G_{strain}}{RT}} \tag{6.9}$$

$$= K_0 \cdot e^{\frac{-\Delta G_{strain}}{RT}} \tag{6.10}$$

$$\tag{6.11}$$

The model assumes that the crystal is elastically isotropic.

References

pyrolite.mineral.lattice.**youngs_modulus_approximation**(*z: int*, *r: float*) → float

Young's modulus approximation for cationic sites in silicates and oxides [#pyrolite.mineral.lattice.youngs_modulus_approximation_1]_ [#pyrolite.mineral.lattice.youngs_modulus_approximation_2]_ [#pyrolite.mineral.lattice.youngs_modulus_approximation_3]_.4

Parameters

- **z** (*int*) – Cationic valence.
- **r** (*float*) – Ionic radius of the cation (Å).

Returns

E – Young's modulus for the cationic site, in Pascals (Pa).

Return type

float

Notes

The bulk modulus K for an an ionic crystal is esimated using [#pyrolite.mineral.lattice.youngs_modulus_approximation_1]_:

$$K = \frac{AZ_aZ_ce^2(n-1)}{9d_0V_0}$$

Where A is the Madelung constant, Z_c and Z_a are the anion and cation valences, e is the charge on the electron, n is the Born power law coefficent, and d_0 is the cation-anion distance [#pyrolite.mineral.lattice.youngs_modulus_approximation_1]_. Using the Shannon ionic radius for oxygen (1.38 Å), this is approximated for cations coordinated by oxygen in silicates and oxides using the following relationship [#pyrolite.mineral.lattice.youngs_modulus_approximation_2]_:

$$K = 750Z_cd^{-3}$$

Where d is the cation-anion distance (Å), Z_c is the cationic valence (and K is in GPa). The Young's modulus E is then calculated through the relationship [#pyrolite.mineral.lattice.youngs_modulus_approximation_3]_:

$$E = 3K(1-2\sigma)$$

Where σ is Poisson's ratio, which in the case of minerals can be approimxated by 0.25 [#pyrolite.mineral.lattice.youngs_modulus_approximation_3]_, and hence:

$$E \approx 1.5K \tag{6.12}$$

$$E \approx 1025Z_cd^{-3} \tag{6.13}$$

References

pyrolite.mineral.lattice.**fit_lattice_strain**(*radii: ndarray*, *ys: ndarray*, *E: float | None = None*, *z: int = 3*, *bounds: list[tuple[float, float]] | None = None*, *r0: float | None = None*, *t0: float = 773.15*, *d0: float = 1.0*, ***kwargs*) → tuple[float, float, float]

Fit a lattice strain model to a given set of abundances.

Parameters

- **radii** (*numpy.ndarray*) – Radii to fit against.
- **ys** (*numpy.ndarray*) – Partition coefficients for given elemental data.
- **E** (*float*) – Young's modulus (stiffness) for the site, in pascals (Pa). Will be estimated using *youngs_modulus_approximation* if none is given.
- **z** (*int*) – Optional specification of cationic valence, for calcuation of approximate Young's modulus using *youngs_modulus_approximation*, where the modulus is not specified.
- **bounds** (*list*) – List of tuples specifying bounds on parameters *ri*, *T* and *D*.

Returns

ri, tk, D – Radius, temperature and partition coefficeint describing the lattice strain fit.

Return type

float

Notes

- Uses the youngs modulus approximation where *E* not provided.
- Passes keyword arguments to *scipy.optimize_curve_fit*.

pyrolite.mineral.mindb

Submodule for accessing the rock forming mineral database.

Notes

Accessing and modifying the database across multiple with multiple threads/processes *could* result in database corruption (e.g. through repeated truncation etc).

pyrolite.mineral.mindb.**list_groups**() → list[str]

List the mineral groups present in the mineral database.

Return type

list

pyrolite.mineral.mindb.**list_minerals**() → list[str]

List the minerals present in the mineral database.

Return type

list

pyrolite.mineral.mindb.**list_formulae**() → list[str]

List the mineral formulae present in the mineral database.

Return type

list

pyrolite.mineral.mindb.**get_mineral**(*name: str = ''*) → Series

Get a specific mineral from the database.

Parameters
name (*str*) – Name of the desired mineral.

Return type
pd.Series

pyrolite.mineral.mindb.**parse_composition**(*composition: str | Formula | Series*, *drop_zeros: bool = True*) → Series

Parse a composition reference to provide an ionic elemental version in the form of a pandas.Series. Currently accepts pandas.Series, *periodictable.formulas.Formula* and structures which will directly convert to pandas.Series (list of tuples, dict).

Parameters

- **composition** (str | *periodictable.formulas.Formula* | pandas.Series) – Name of a mineral, a formula or composition as a series
- **drop_zeros** (*bool*) – Whether to drop compositional zeros.

Returns
mineral – Composition formatted as a series.

Return type
pandas.Series

pyrolite.mineral.mindb.**get_mineral_group**(*group: str = ''*) → DataFrame

Extract a mineral group from the database.

Parameters
group (*str*) – Group to extract from the mineral database.

Returns
Dataframe of group members and compositions.

Return type
pandas.DataFrame

pyrolite.mineral.mindb.**update_database**(*path: str | Path | None = None*, ***kwargs*)

Update the mineral composition database from the CSV.`

Parameters
path (*str | pathlib.Path*) – The desired filepath for the JSON database.

Notes

This will take the 'mins.csv' file from the mineral pyrolite data folder and construct a document-based JSON database.

6.1.5 pyrolite.util

Various utilities used by other submodules.

pyrolite.util.general

class pyrolite.util.general.**Timewith**(*name: str = ''*)

property elapsed: `float`

checkpoint(*name: str = ''*)

pyrolite.util.general.**temp_path**(*suffix: str = ''*) → Path

Return the path of a temporary directory.

pyrolite.util.general.**tempdir**(***kwargs*)

pyrolite.util.general.**flatten_dict**(*d: dict*, *climb: bool = False*, *safemode: bool = False*) → dict

Flattens a nested dictionary containing only string keys.

This will work for dictionaries which don't have two equivalent keys at the same level. If you're worried about this, use safemode=True.

Partially taken from https://stackoverflow.com/a/6043835.

Parameters

- **climb** (bool, *False*) – Whether to keep trunk or leaf-values, for items with the same key.
- **safemode** (bool, *True*) – Whether to keep all keys as a tuple index, to avoid issues with conflicts.

Returns
Flattened dictionary.

Return type
dict

pyrolite.util.general.**swap_item**(*startlist: list*, *pull: Any*, *push: Any*) → list

Swap a specified item in a list for another.

Parameters

- **startlist** (*list*) – List to replace item within.
- **pull** – Item to replace in the list.
- **push** – Item to add into the list.

Return type
list

pyrolite.util.general.**copy_file**(*src: str | Path*, *dst: str | Path*, *ext: str | None = None*, *permissions: str | None = None*)

Copy a file from one place to another. Uses the full filepath including name.

Parameters

- **src** (*str | pathlib.Path*) – Source filepath.
- **dst** (*str | pathlib.Path*) – Destination filepath or directory.
- **ext** (str, *None*) – Optional file extension specification.

pyrolite.util.general.**remove_directory**(*directory: Path*)

Remove a specific directory, contained files and sub-directories.

Parameters
directory (*str, Path*) – Path to directory.

pyrolite.util.pd

pyrolite.util.pd.**drop_where_all_empty**(*df: DataFrame*) → DataFrame

Drop rows and columns which are completely empty.

Parameters
df (*pandas.DataFrame | pandas.Series*) – Pandas object to ensure is in the form of a series.

pyrolite.util.pd.**read_table**(*filepath: str | Path*, *index_col: int = 0*, ***kwargs*) → DataFrame

Read tabluar data from an excel or csv text-based file.

Parameters
filepath (*str | pathlib.Path*) – Path to file.

Return type
pandas.DataFrame

pyrolite.util.pd.**column_ordered_append**(*df1: DataFrame*, *df2: DataFrame*, ***kwargs*) → DataFrame

Appends one dataframe to another, preserving the column order of the first and adding new columns on the right. Also accepts and passes on standard keyword arguments for pd.DataFrame.append.

Parameters

- **df1** (*pandas.DataFrame*) – The dataframe for which columns order is preserved in the output.
- **df2** (*pandas.DataFrame*) – The dataframe for which new columns are appended to the output.

Return type
pandas.DataFrame

pyrolite.util.pd.**accumulate**(*dfs: list[DataFrame]*, *ignore_index: bool = False*, *trace_source: bool = False*, *names: list[str] | None = None*)

Accumulate an iterable containing multiple pandas.DataFrame to a single frame.

Parameters

- **dfs** (*list*) – Sequence of dataframes.
- **ignore_index** (*bool*) – Whether to ignore the indexes upon joining.
- **trace_source** (*bool*) – Whether to retain a reference to the source of the data rows.
- **names** (*list*) – Names to use in place of indexes for source names.

Returns
Accumulated dataframe.

Return type
pandas.DataFrame

pyrolite.util.pd.**to_frame**(*ser: Series | DataFrame*) → DataFrame

Simple utility for converting to pandas.DataFrame.

Parameters
ser (*pandas.Series | pandas.DataFrame*) – Pandas object to ensure is in the form of a dataframe.

Return type
pandas.DataFrame

pyrolite.util.pd.**to_ser**(*df: DataFrame | Series*) → Series

Simple utility for converting single column pandas.DataFrame to pandas.Series.

Parameters
df (*pandas.DataFrame | pandas.Series*) – Pandas object to ensure is in the form of a series.

Return type
pandas.Series

pyrolite.util.pd.**to_numeric**(*df: DataFrame | Series*, *errors: str = 'coerce'*, *exclude: list[str] | None = None*) → DataFrame | Series

Converts non-numeric columns to numeric type where possible.

Notes

Avoid using .loc or .iloc on the LHS to make sure that data dtypes are propagated.

pyrolite.util.pd.**zero_to_nan**(*df: DataFrame*, *rtol: float = 1e-05*, *atol: float = 1e-08*) → DataFrame

Replace floats close, less or equal to zero with np.nan in a dataframe.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to censor.
- **rtol** (*float*) – The relative tolerance parameter.
- **atol** (*float*) – The absolute tolerance parameter.

Returns
Censored DataFrame.

Return type
pandas.DataFrame

pyrolite.util.pd.**outliers**(*df: DataFrame*, *cols: list[str] | Index | None = None*, *detect: Callable = <function <lambda>>*, *quantile_select: tuple[float, float]=(0.02, 0.98)*, *logquantile: bool = False*, *exclude: bool = False*) → DataFrame

pyrolite.util.pd.**concat_columns**(*df: DataFrame*, *columns: list[str] | None = None*, *astype: type | dtype = <class 'str'>*, ***kwargs*) → Series

Concatenate strings across columns.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to concatenate.
- **columns** (*list*) – List of columns to concatenate.
- **astype** (*type*) – Type to convert final concatenation to.

Return type
pandas.Series

pyrolite.util.pd.**uniques_from_concat**(*df: DataFrame*, *columns: list[str] | None = None*, *hashit: bool = True*) → Series

Creates ideally unique keys from multiple columns. Optionally hashes string to standardise length of identifier.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to create indexes for.
- **columns** (*list*) – Columns to use in the string concatenatation.
- **hashit** (bool, *True*) – Whether to use a hashing algorithm to create the key from a typically longer string.

Return type
pandas.Series

pyrolite.util.pd.**df_from_csvs**(*csvs: list[str | Path]*, *dropna: bool = True*, *ignore_index: bool = False*, ***kwargs*)

Takes a list of .csv filenames and converts to a single DataFrame. Combines columns across dataframes, preserving order of the first entered.

E.g. SiO2, Al2O3, MgO, MnO, CaO SiO2, MgO, FeO, CaO SiO2, Na2O, Al2O3, FeO, CaO => SiO2, Na2O, Al2O3, MgO, FeO, MnO, CaO - Existing neighbours take priority (i.e. FeO won't be inserted bf Al2O3) - Earlier inputs take priority (where ordering is ambiguous, place the earlier first)

Todo

Attempt to preserve column ordering across column sets, assuming they are generally in the same order but preserving only some of the information.

pyrolite.util.plot

Utility functions for working with matplotlib.

Parameters

- **DEFAULT_CONT_COLORMAP** (*matplotlib.colors.ScalarMappable*) – Default continuous colormap.
- **DEFAULT_DICS_COLORMAP** (*matplotlib.colors.ScalarMappable*) – Default discrete colormap.
- **USE_PCOLOR** (*bool*) – Option to use the `matplotlib.pyplot.pcolor()` function in place of `matplotlib.pyplot.pcolormesh()`.

pyrolite.util.plot.axes

Functions for creating, ordering and modifying `Axes`.

pyrolite.util.plot.axes.**get_ordered_axes**(*fig: Figure*) → list[Axes] | ndarray

Get the axes from a figure, which may or may not have been modified by pyrolite functions. This ensures that ordering is preserved.

pyrolite.util.plot.axes.**get_axes_index**(*ax: Axes*) → tuple[int, int, int]

Get the three-digit integer index of a subplot in a regular grid.

Parameters
ax (*matplotlib.axes.Axes*) – Axis to to get the gridspec index for.

Returns
Rows, columns and axis index for the gridspec.

Return type
tuple

pyrolite.util.plot.axes.**replace_with_ternary_axis**(*ax: Axes*) → TernaryAxes

Replace a specified axis with a ternary equivalent.

Parameters
ax (*matplotlib.axes.Axes*)

Returns
tax

Return type
mpltern.ternary.TernaryAxes

pyrolite.util.plot.axes.**label_axes**(*ax: Axes*, *labels: list[str] | None = None*, ***kwargs*)

Convenience function for labelling rectilinear and ternary axes.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to label.
- **labels** (*list*) – List of labels: [x, y] | or [t, l, r]

pyrolite.util.plot.axes.**axes_to_ternary**(*ax: Axes*) → list[Axes]

Set axes to ternary projection after axis creation. As currently implemented, note that this will replace and reorder axes as acecessed from the figure (the ternary axis will then be at the end), and as such this returns a list of axes in the correct order.

Parameters
ax (*matplotlib.axes.Axes | list (matplotlib.axes.Axes)*) – Axis (or axes) to convert projection for.

Returns
axes

Return type
list' (matplotlib.axes.Axes, class: `~mpltern.ternary.TernaryAxes)

pyrolite.util.plot.axes.**check_default_axes**(*ax: Axes*) → bool

Simple test to check whether an axis is empty of artists and hasn't been rescaled from the default extent.

Parameters
ax (*matplotlib.axes.Axes*) – Axes to check for artists and scaling.

Return type
bool

pyrolite.util.plot.axes.**check_empty**(*ax: Axes*) → bool

Simple test to check whether an axis is empty of artists.

Parameters
ax (*matplotlib.axes.Axes*) – Axes to check for artists.

Return type
bool

pyrolite.util.plot.axes.**init_axes**(*ax: Axes | None = None*, *projection: str | None = None*, *minsize: float = 1.0*, *xscale: Literal['linear', 'log'] = 'linear'*, *yscale: Literal['linear', 'log'] = 'linear'*, ***kwargs*) → Axes | list[Axes] | ndarray

Get or create an Axes from an optionally-specified starting Axes.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Specified starting axes, optional.
- **projection** (*str*) – Whether to create a projected (e.g. ternary) axes.
- **minsize** (*float*) – Minimum figure dimension (inches).
- **xscale** (*str*) – Scale for the x-axis - 'linear' or 'log'.
- **yscale** (*str*) – Scale for the y-axis - 'linear' or 'log'.

Returns
ax

Return type
matplotlib.axes.Axes

pyrolite.util.plot.axes.**share_axes**(*axes: list[Axes] | ndarray*, *which: tuple[Literal['x', 'y']] | Literal['x', 'y', 'xy', 'yx', 'both'] = 'xy'*)

Link the x, y or both axes across a group of matplotlib.axes.Axes.

Parameters

- **axes** (*list*) – List of axes to link.
- **which** (*str*) – Which axes to link. If *x*, link the x-axes; if *y* link the y-axes, otherwise link both.

pyrolite.util.plot.axes.**get_twins**(*ax: Axes*, *which: tuple[Literal['x', 'y']] | Literal['x', 'y', 'xy'] = 'y'*) → list[Axes]

Get twin axes of a specified axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to get twins for.
- **which** (*str*) – Which twins to get (shared *'x'*, shared *'y'* or the concatenatation of both, *'xy'*).

Return type
list

Notes

This function was designed to assist in avoiding creating a series of duplicate axes when replotting on an existing axis using a function which would typically create a twin axis.

pyrolite.util.plot.axes.**subaxes**(*ax: Axes*, *side: Literal['bottom', 'top', 'left', 'right'] = 'bottom'*, *width: float = 0.2*, *moveticks: bool = True*) → Axes

Append a sub-axes to one side of an axes.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes to append a sub-axes to.
- **side** (*str*) – Which side to append the axes on.
- **width** (*float*) – Fraction of width to give to the subaxes.
- **moveticks** (*bool*) – Whether to move ticks to the outer axes.

Returns
Subaxes instance.

Return type
matplotlib.axes.Axes

pyrolite.util.plot.axes.**add_colorbar**(*mappable*, ***kwargs*) → Colorbar

Adds a colorbar to a given mappable object.

Source: http://joseph-long.com/writing/colorbars/

Parameters
mappable – The Image, ContourSet, etc. to which the colorbar applies.

Return type
matplotlib.colorbar.Colorbar

Todo

- Where no mappable specificed, get most recent axes, and check for collections etc

pyrolite.util.plot.density

Functions for dealing with kernel density estimation.

ivar USE_PCOLOR
Option to use the *matplotlib.pyplot.pcolor* function in place of *matplotlib.pyplot.pcolormesh*.

vartype USE_PCOLOR
bool

pyrolite.util.plot.density.**get_axis_density_methods**(*ax: Axes*) → tuple[Callable, Callable, Callable]

Get the relevant density and contouring methods for a given axis.

Parameters
ax (matplotlib.axes.Axes | *mpltern.ternary.TernaryAxes*) – Axis to check.

Returns
Relevant functions for this axis.

Return type
pcolor, contour, contourf

pyrolite.util.plot.density.**percentile_contour_values_from_meshz**(*z: ndarray[tuple[int, int], dtype[number]]*, *percentiles: list[float] | None = None*, *resolution: int = 1000*) → tuple[list[float | str], list[float]]

Integrate a probability density distribution Z(X,Y) to obtain contours in Z which correspond to specified percentile contours. Contour values will be returned with the same order as the inputs.

Parameters

- **z** (*numpy.ndarray*) – Probability density function over x, y.
- **percentiles** (*numpy.ndarray*) – Percentile values for which to create contours.
- **resolution** (*int*) – Number of bins for thresholds between 0. and max(Z)

Returns

- **labels** (*list*) – Labels for contours (percentiles, if above minimum z value).
- **contours** (*list*) – Contour height values.

Todo

This may error for a list of percentiles where one or more requested values are below the miniumum threshold. The exception handling should be updated to cater for arrays - where some of the values may be above the minimum.

pyrolite.util.plot.density.**plot_Z_percentiles**(**coords: ndarray[tuple[int]], zi: ndarray[tuple[int, int]], percentiles: list[float] | None = None, ax: Axes | None = None, extent: list[float] | None = None, fontsize: float = 8, cmap: Colormap | str | None = None, colors: str | list[str] | list[tuple[float, float, float, float]] | None = None, linewidths: float | list[float] | None = None, linestyles: list[str] | None = None, contour_labels: list[str] | None = None, label_contours: bool = True, **kwargs*) → QuadContourSet

Plot percentile contours onto a 2D (scaled or unscaled) probability density distribution Z over X,Y.

Parameters

- **coords** (*numpy.ndarray*) – Arrays of (x, y) or (a, b, c) coordinates.
- **z** (*numpy.ndarray*) – Probability density function over x, y.
- **percentiles** (*list*) – Percentile values for which to create contours.
- **ax** (matplotlib.axes.Axes, *None*) – Axes on which to plot. If none given, will create a new Axes instance.
- **extent** (list, *None*) – List or np.ndarray in the form [-x, +x, -y, +y] over which the image extends.
- **fontsize** (*float*) – Fontsize for the contour labels.
- **cmap** (*matplotlib.colors.ListedColormap*) – Color map for the contours and contour labels.
- **colors** (*str | list*) – Colors for the contours, can optionally be specified *in place of cmap.*
- **linewidths** (*str | list*) – Widths of contour lines.
- **linestyles** (*str | list*) – Styles for contour lines.
- **contour_labels** (*dict | list*) – Labels to assign to contours, organised by level.
- **label_contours bool** – Whether to add text labels to individual contours.

Returns

Plotted and formatted contour set.

Return type

matplotlib.contour.QuadContourSet

Notes

When the contours are percentile based, high percentile contours tend to get washed our with colormapping - consider adding different controls on coloring, especially where there are only one or two contours specified. One way to do this would be via the string based keyword argument *colors* for plt.contour, with an adaption for non-string colours which post-hoc modifies the contour lines based on the specified colours?

pyrolite.util.plot.density.**conditional_prob_density**(*y: ndarray[tuple[int, int]], x: ndarray[tuple[int]] | None = None, logy: bool = False, resolution: int = 5, bins: int = 50, yextent: tuple[float, float] | None = None, rescale: bool = True, mode: str = 'binkde', ret_centres: bool = False, **kwargs*) → tuple[ndarray[tuple[int]], ndarray[tuple[int]], ndarray[tuple[int, int]]] | tuple[ndarray[tuple[int]], ndarray[tuple[int]], ndarray[tuple[int]], ndarray[tuple[int]], ndarray[tuple[int, int]]]

Estimate the conditional probability density of one dependent variable.

Parameters

- **y** (*numpy.ndarray*) – Dependent variable for which to calculate conditional probability P(y | X=x)
- **x** (numpy.ndarray, *None*) – Optionally-specified independent index.
- **logy** (*bool*) – Whether to use a logarithmic bin spacing on the y axis.
- **resolution** (*int*) – Points added per segment via interpolation along the x axis.
- **bins** (*int*) – Bins for histograms and grids along the independent axis.
- **yextent** (*tuple*) – Extent in the y direction.
- **rescale** (*bool*) – Whether to rescale bins to give the same max Z across x.
- **mode** (*str*) –

  Mode of computation.

  If mode is *"ckde"*, use *statsmodels.nonparametric.KDEMultivariateConditional* to compute a conditional kernel density estimate. If mode is *"kde"*, use a normal gaussian kernel density estimate. If mode is *"binkde"*, use a gaussian kernel density estimate over y for each bin. If mode is *"hist"*, compute a histogram.
- **ret_centres** (*bool*) – Whether to return bin centres in addtion to histogram edges, e.g. for later contouring.

Returns

x bin edges *xe*, *y* bin edges *ye*, histogram/density estimates *Z*. If *ret_centres* is *True*, the last two return values will contain the bin centres *xi*, *yi*.

Return type

tuple of numpy.ndarray

pyrolite.util.plot.export

Functions for export of figures and figure elements from matplolib.

`pyrolite.util.plot.export.`**`save_figure`**(*figure: Figure*, *name: str = 'fig'*, *save_at: str | Path = ''*, *save_fmts: list[str] | None = None*, ***kwargs*)

Save a figure at a specified location in a number of formats.

`pyrolite.util.plot.export.`**`save_axes`**(*ax: Axes*, *name: str = 'fig'*, *save_at: str | Path = ''*, *save_fmts: list[str] | None = None*, *pad: float = 0.0*, ***kwargs*)

Save either a single or multiple axes (from a single figure) based on their extent. Uses the save_figure procedure to save at a specific location using a number of formats.

Todo

- Add legend to items

`pyrolite.util.plot.export.`**`get_full_extent`**(*ax: Axes*, *pad: float = 0.0*) → Bbox

Get the full extent of an axes, including axes labels, tick labels, and titles. Text objects are first drawn to define the extents.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axes of which to check items to get full extent.
- **pad** (*float | tuple*) – Amount of padding to add to the full extent prior to returning. If a tuple is passed, the padding will be as above, but for x and y directions, respectively.

Returns
Bbox of the axes with optional additional padding.

Return type
matplotlib.transforms.Bbox

pyrolite.util.plot.export.**path_to_csv**(*path: Path | tuple*, *xname: str = 'x'*, *yname: str = 'y'*, *delim: str = ','*, *linesep: str = '\n'*) → str

Extract the verticies from a path and write them to csv.

Parameters

- **path** (*matplotlib.path.Path | tuple*) – Path or x-y tuple to use for coordinates.
- **xname** (*str*) – Name of the x variable.
- **yname** (*str*) – Name of the y variable.
- **delim** (*str*) – Delimiter for the csv file.
- **linesep** (*str*) – Line separator character.

Returns
String-representation of csv file, ready to be written to disk.

Return type
str

pyrolite.util.plot.grid

Gridding and binning functions.

pyrolite.util.plot.grid.**bin_centres_to_edges**(*centres: ndarray*, *sort: bool = True*) → ndarray

Translates point estimates at the centres of bins to equivalent edges, for the case of evenly spaced bins.

Todo

- This can be updated to unevenly spaced bins, just need to calculate outer bins.

pyrolite.util.plot.grid.**bin_edges_to_centres**(*edges: ndarray*) → ndarray

Translates edges of histogram bins to bin centres.

pyrolite.util.plot.grid.**ternary_grid**(*data: ndarray | None = None*, *nbins: int = 10*, *margin: float = 0.001*, *force_margin: bool = False*, *yscale: float = 1.0*, *tfm: Callable = <function <lambda>>*) → tuple[ndarray, ndarray, tuple[ndarray, ndarray], tuple[ndarray, ndarray]]

Construct a graphical linearly-spaced grid within a ternary space.

Parameters

- **data** (*numpy.ndarray*) – Data to construct the grid around (*(samples, 3)*).
- **nbins** (*int*) – Number of bins for grid.

- **margin** (*float*) – Proportional value for the position of the outer boundary of the grid.
- **forge_margin** (*bool*) – Whether to enforce the grid margin.
- **yscale** (*float*) – Y scale for the specific ternary diagram.
- **tfm** – Log transform to use for the grid creation.

Returns

- **bins** (*numpy.ndarray*) – Bin centres along each of the ternary axes (*(samples, 3)*)
- **binedges** (*numpy.ndarray*) – Position of bin edges.
- **centregrid** (*list of numpy.ndarray*) – Meshgrid of bin centres.
- **edgegrid** (*list of numpy.ndarray*) – Meshgrid of bin edges.

pyrolite.util.plot.helpers

matplotlib helper functions for commong drawing tasks.

pyrolite.util.plot.helpers.**alphalabel_subplots**(*ax: Axes*, *fmt: str = '{}'*, *xy: tuple[float, float] = (0.03, 0.95)*, *ha: str = 'left'*, *va: str = 'top'*, ***kwargs*)

Add alphabetical labels to a successive series of subplots with a specified format.

Parameters

- **ax** (*list | numpy.ndarray | numpy.flatiter*) – Axes to label, in desired order.
- **fmt** (*str*) – Format string to use. To add e.g. parentheses, you could specify *"({})"*.
- **xy** (*tuple*) – Position of the labels in axes coordinates.
- **ha** (*str*) – Horizontal alignment of the labels (*{"left", "right"}*).
- **va** (*str*) – Vertical alignment of the labels (*{"top", "bottom"}*).

pyrolite.util.plot.helpers.**get_centroid**(*poly: Polygon*) → tuple[float, float]

Centroid of a closed polygon using the Shoelace formula.

Parameters
poly (*matplotlib.patches.Polygon*) – Polygon to obtain the centroid of.

Returns
cx, cy – Centroid coordinates.

Return type
tuple

pyrolite.util.plot.helpers.**get_visual_center**(*poly: Polygon*, *vertical_exaggeration: float = 1*) → tuple[float, float]

Visual center of a closed polygon.

Parameters

- **poly** (*matplotlib.patches.Polygon*) – Polygon for which to obtain the visual center.
- **vertical_exaggeration** (*float*) – Apparent vertical exaggeration of the plot (pixels per unit in y direction divided by pixels per unit in the x direction).

Returns
cx, cy – Centroid coordinates.

Return type
tuple

pyrolite.util.plot.helpers.**rect_from_centre**(*x: float*, *y: float*, *dx: float = 0*, *dy: float = 0*, ***kwargs*) → Rectangle

Takes an xy point, and creates a rectangular patch centred about it.

pyrolite.util.plot.helpers.**draw_vector**(*v0: ndarray*, *v1: ndarray*, *ax: Axes | None = None*, ***kwargs*)

Plots an arrow represnting the direction and magnitue of a principal component on a biaxial plot.

Modified after Jake VanderPlas' Python Data Science Handbook https://jakevdp.github.io/PythonDataScienceHandbook/ 05.09-principal-component-analysis.html

Todo

Update for ternary plots.

pyrolite.util.plot.helpers.**vector_to_line**(*mu: ndarray*, *vector: ndarray*, *variance: float*, *spans: int = 4*, *expand: int = 10*) → ndarray

Creates an array of points representing a line along a vector - typically for principal component analysis. Modified after Jake VanderPlas' Python Data Science Handbook https://jakevdp.github.io/PythonDataScienceHandbook/ 05.09-principal-component-analysis.html

pyrolite.util.plot.helpers.**plot_stdev_ellipses**(*comp: ndarray*, *nstds: float = 4*, *scale: float = 100*, *resolution: int = 1000*, *transform: Callable | None = None*, *ax: Axes | None = None*, ***kwargs*) → Axes

Plot covariance ellipses at a number of standard deviations from the mean.

Parameters

- **comp** (*numpy.ndarray*) – Composition to use.
- **nstds** (*int*) – Number of standard deviations from the mean for which to plot the ellipses.
- **scale** (*float*) – Scale applying to all x-y data points. For intergration with python-ternary.
- **transform** (*Callable*) – Function for transformation of data prior to plotting (to either 2D or 3D).
- **ax** (*matplotlib.axes.Axes*) – Axes to plot on.

Returns

ax

Return type

matplotlib.axes.Axes

pyrolite.util.plot.helpers.**plot_2dhull**(*data: ndarray*, *ax: Axes | None = None*, *splines: bool = False*, *s: int = 0*, ***plotkwargs*) → list[Line2D]

Plots a 2D convex hull around an array of xy data points.

pyrolite.util.plot.helpers.**plot_cooccurence**(*arr: ndarray*, *ax: Axes | None = None*, *normalize: bool = True*, *log: bool = False*, *colorbar: bool = False*, ***kwargs*) → Axes

Plot the co-occurence frequency matrix for a given input.

Parameters

- **ax** (matplotlib.axes.Axes, *None*) – The subplot to draw on.
- **normalize** (*bool*) – Whether to normalize the cooccurence to compare disparate variables.
- **log** (*bool*) – Whether to take the log of the cooccurence.

- **colorbar** (*bool*) – Whether to append a colorbar.

Returns
Axes on which the cooccurence plot is added.

Return type
matplotlib.axes.Axes

pyrolite.util.plot.helpers.**nan_scatter**(*xdata: ndarray*, *ydata: ndarray*, *ax: Axes | None = None*, *axes_width: float = 0.2*, ***kwargs*) → Axes

Scatter plot with additional marginal axes to plot data for which data is partially missing. Additional keyword arguments are passed to matplotlib.

Parameters

- **xdata** (*numpy.ndarray*) – X data
- **ydata** (*class:numpy.ndarray | pd.Series*) – Y data
- **ax** (*matplotlib.axes.Axes*) – Axes on which to plot.
- **axes_width** (*float*) – Width of the marginal axes.

Returns
Axes on which the nan_scatter is plotted.

Return type
matplotlib.axes.Axes

pyrolite.util.plot.helpers.**init_spherical_octant**(*angle_indicated: float = 30*, *labels: list[str] = None*, *view_init: tuple[float, float] = (25, 55)*, *fontsize: float = 10*, ***kwargs*) → Axes

Initalize a figure with a 3D octant of a unit sphere, appropriately labeled with regard to angles corresponding to the handling of the respective compositional data transformation function (*~pyrolite.comp.codata.sphere*).

Parameters

- **angle_indicated** (*float*) – Angle relative to axes for indicating the relative positioning, optional.
- **labels** (*list*) – Optional specification of data/axes labels. This will be used for labelling of both the axes and optionally-added arcs specifying which angles are represented.

Returns
ax – Initialized 3D axis.

Return type
matplotlib.axes.Axes3D

pyrolite.util.plot.helpers.**plot_pca_vectors**(*comp: ndarray*, *nstds: float = 2*, *scale: float = 100.0*, *transform: Callable | None = None*, *ax: Axes | None = None*, *colors: str | list[str] | None = None*, *linestyles: str | list[str] | None = None*, ***kwargs*) → Axes

Plot vectors corresponding to principal components and their magnitudes.

Parameters

- **comp** (*numpy.ndarray*) – Composition to use.
- **nstds** (*int*) – Multiplier for magnitude of individual principal component vectors.
- **scale** (*float*) – Scale applying to all x-y data points. For intergration with python-ternary.

- **transform** (*Callable*) – Function for transformation of data prior to plotting (to either 2D or 3D).
- **ax** (*matplotlib.axes.Axes*) – Axes to plot on.

Returns
ax

Return type
matplotlib.axes.Axes

Todo

- Minor reimplementation of the sklearn PCA to avoid dependency.

  https://en.wikipedia.org/wiki/Principal_component_analysis

pyrolite.util.plot.interpolation

Line interpolation for matplotlib lines and paths.

pyrolite.util.plot.interpolation.**interpolate_path**(*path: Path*, *resolution: int = 100*, *periodic: bool = False*, *aspath: bool = True*, *closefirst: bool = False*, ***kwargs*) → Path | ndarray

Obtain the interpolation of an existing path at a given resolution. Keyword arguments are forwarded to `scipy.interpolate.splprep()`.

Parameters

- **path** (*matplotlib.path.Path*) – Path to interpolate.
- **resolution int** – Resolution at which to obtain the new path. The verticies of the new path will have shape (*resolution*, 2).
- **periodic** (*bool*) – Whether to use a periodic spline.
- **periodic** (*bool*) – Whether to return a matplotlib.path.Path, or simply a tuple of x-y arrays.
- **closefirst** (*bool*) – Whether to first close the path by appending the first point again.

Returns
Interpolated path object, if *aspath* is *True*, else a tuple of x-y arrays.

Return type
matplotlib.path.Path | tuple

pyrolite.util.plot.interpolation.**interpolated_patch_path**(*patch: Patch*, *resolution: int = 100*, ***kwargs*) → Path | ndarray

Obtain the periodic interpolation of the existing path of a patch at a given resolution.

Parameters

- **patch** (*matplotlib.patches.Patch*) – Patch to obtain the original path from.
- **resolution int** – Resolution at which to obtain the new path. The verticies of the new path will have shape (*resolution*, 2).

Returns
Interpolated path object.`

Return type
matplotlib.path.Path

pyrolite.util.plot.interpolation.**get_contour_paths**(*src: Axes | QuadContourSet*, *resolution: int = 100*, *minsize: int = 3*, *filter: bool = True*) → tuple[list[ndarray], list[str], list[dict]]

Extract the paths of contours from a contour plot.

Parameters

- **ax** (matplotlib.axes.Axes | *matplotlib.contour.QuadContourSet*) – Axes to extract contours from.
- **resolution** (*int*) – Resolution of interpolated splines to return.
- **filter** (*bool*) – Whether to filter out paths which have no length.

Returns

- **contourspaths** (*list (list)*) – List of lists, each represnting one line collection (a single contour). In the case where this contour is multimodal, there will be multiple paths for each contour.
- **contournames** (*list*) – List of names for contours, where they have been labelled, and there are no other text artists on the figure.
- **contourstyles** (*list*) – List of styles for contours.

pyrolite.util.plot.legend

Functions for creating and modifying legend entries for matplotlib.

Todo

- Functions for working with and modifying legend entries. ax.lines + ax.patches + ax.collections + ax.containers, handle ax.parasites

pyrolite.util.plot.legend.**proxy_rect**(***kwargs*) → Rectangle

Generates a legend proxy for a filled region.

Return type
matplotlib.patches.Rectangle

pyrolite.util.plot.legend.**proxy_line**(***kwargs*) → Line2D

Generates a legend proxy for a line region.

Return type
matplotlib.lines.Line2D

pyrolite.util.plot.legend.**modify_legend_handles**(*ax: Axes*, ***kwargs*) → tuple[list, list[str]]

Modify the handles of a legend based for a single axis.

Parameters
ax (*matplotlib.axes.Axes*) – Axis for which to obtain modifed legend handles.

Returns

- **handles** (*list*) – Handles to be passed to a legend call.
- **labels** (*list*) – Labels to be passed to a legend call.

pyrolite.util.plot.style

Functions for automated plot styling and argument handling.

ivar DEFAULT_CONT_COLORMAP
Default continuous colormap.

vartype DEFAULT_CONT_COLORMAP
matplotlib.colors.ScalarMappable

ivar DEFAULT_DISC_COLORMAP
Default discrete colormap.

vartype DEFAULT_DISC_COLORMAP
matplotlib.colors.ScalarMappable

pyrolite.util.plot.style.**linekwargs**(*kwargs: dict*) → dict
Get a subset of keyword arguments to pass to a matplotlib line-plot call.

Parameters
kwargs (*dict*) – Dictionary of keyword arguments to subset.

Return type
dict

pyrolite.util.plot.style.**scatterkwargs**(*kwargs: dict*) → dict
Get a subset of keyword arguments to pass to a matplotlib scatter call.

Parameters
kwargs (*dict*) – Dictionary of keyword arguments to subset.

Return type
dict

pyrolite.util.plot.style.**patchkwargs**(*kwargs: dict*) → dict

pyrolite.util.plot.style.**marker_cycle**(*markers: list[str] | None = None*) → cycle
Cycle through a set of markers.

Parameters
markers (*list*) – List of markers to provide to matplotlib.

pyrolite.util.plot.style.**mappable_from_values**(*values: ~numpy.ndarray | ~pandas.Series | list[~numpy.number], cmap: str | ~matplotlib.colors.Colormap | None = <matplotlib.colors.ListedColormap object>, norm: ~matplotlib.colors.Norm | None = None, **kwargs*) → _ScalarMappable
Create a scalar mappable object from an array of values.

Return type
matplotlib.cm.ScalarMappable

pyrolite.util.plot.style.**ternary_color**(*tlr: ndarray[tuple[int, int]], alpha: float = 1.0, colors: tuple[str | list[float], ...] = ([1, 0, 0], [0, 1, 0], [0, 0, 1]), coefficients: tuple[float, float, float] = (1, 1, 1)*) → ndarray[tuple[int, int]]
Color a set of points by their ternary combinations of three specified colors.

Parameters

- **tlr** (*numpy.ndarray*)

- **alpha** (*float*) – Alpha coefficient for the colors; to be applied *multiplicatively* with any existing alpha value (for RGBA colors specified).
- **colors** (*tuple*) – Set of colours corresponding to the top, left and right verticies, respectively.
- **coefficients** (*tuple*) – Coefficients for the ternary data to adjust the centre.

Returns
Color array for the ternary points.

Return type
numpy.ndarray

pyrolite.util.plot.style.**color_ternary_polygons_by_centroid**(*ax: Axes | None = None*, *patches: list[Patch] | None = None*, *alpha: float = 1.0*, *colors: tuple[str | list[float], ...] = ([1, 0, 0], [0, 1, 0], [0, 0, 1])*, *coefficients: tuple[float, float, float] = (1, 1, 1)*) → list[Patch]

Color a set of polygons within a ternary diagram by their centroid colors.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Ternary axes to check for patches, if patches is not supplied.
- **patches** (*list*) – List of ternary-hosted patches to apply color to.
- **alpha** (*float*) – Alpha coefficient for the colors; to be applied *multiplicatively* with any existing alpha value (for RGBA colors specified).
- **colors** (*tuple*) – Set of colours corresponding to the top, left and right verticies, respectively.
- **coefficients** (*tuple*) – Coefficients for the ternary data to adjust the centre.

Returns
patches – List of patches, with updated facecolors.

Return type
list

pyrolite.util.plot.transform

Transformation utilites for matplotlib.

pyrolite.util.plot.transform.**affine_transform**(*mtx: ndarray | None = None*) → Callable

Construct a function which will perform a 2D affine transform based on a 3x3 affine matrix.

Parameters
mtx (*numpy.ndarray*)

pyrolite.util.plot.transform.**tlr_to_xy**(*tlr: ndarray*) → ndarray

Transform a ternary coordinate system (top-left-right) to an xy-cartesian coordinate system.

Parameters
tlr (*numpy.ndarray*) – Array of shape (n, 3) in the t-l-r coordinate system.

Returns
xy – Array of shape (n, 2) in the x-y coordinate system.

Return type
numpy.ndarray

pyrolite.util.plot.transform.**xy_to_tlr**(*xy: ndarray*) → ndarray

Parameters
xy (*numpy.ndarray*) – Array of shape (n, 2) in the x-y coordinate system.

Returns
tlr – Array of shape (n, 3) in the t-l-r coordinate system.

Return type
numpy.ndarray

pyrolite.util.plot.transform.**ABC_to_xy**(*ABC: ndarray*, *xscale: float = 1.0*, *yscale: float = 1.0*) → ndarray

Convert ternary compositional coordiantes to x-y coordinates for visualisation within a triangle.

Parameters

- **ABC** (*numpy.ndarray*) – Ternary array (*samples, 3*).
- **xscale** (*float*) – Scale for x-axis.
- **yscale** (*float*) – Scale for y-axis.

Returns
Array of x-y coordinates (*samples, 2*)

Return type
numpy.ndarray

pyrolite.util.plot.transform.**xy_to_ABC**(*xy: ndarray*, *xscale: float = 1.0*, *yscale: float = 1.0*) → ndarray

Convert x-y coordinates within a triangle to compositional ternary coordinates.

Parameters

- **xy** (*numpy.ndarray*) – XY array (*samples, 2*).
- **xscale** (*float*) – Scale for x-axis.
- **yscale** (*float*) – Scale for y-axis.

Returns
Array of ternary coordinates (*samples, 3*)

Return type
numpy.ndarray

pyrolite.util.text

pyrolite.util.text.**to_width**(*multiline_string: str*, *width: int = 79*, ***kwargs*) → str

Uses builtin textwapr for text wrapping to a specific width.

pyrolite.util.text.**normalise_whitespace**(*strg: str*) → str

Substitutes extra tabs, newlines etc. for a single space.

pyrolite.util.text.**remove_prefix**(*z: str*, *prefix: str*) → str

Remove a specific prefix from the start of a string.

pyrolite.util.text.**remove_suffix**(*x: str*, *suffix: str = ' '*) → str

Remove a specific suffix from the end of a string.

pyrolite.util.text.**quoted_string**(*s: str*) → str

pyrolite.util.text.**titlecase**(*s: str*, *exceptions: list[str] | None = None*, *abbrv: list[str] | None = None*, *capitalize_first: bool = True*, *split_on: str = '[\\.\\s_-]+'*, *delim: str = ''*) → str

Formats strings in CamelCase, with exceptions for simple articles and omitted abbreviations which retain their capitalization.

Todo

- Option for retaining original CamelCase.

pyrolite.util.text.**string_variations**(*names: list[str]*, *preprocess: list[str] | None = None*, *swaps: list[tuple[str, str]] | None = None*)

Returns equilvaent string variations based on an input set of strings.

Parameters

- **names** (*list[str]*) – String or list of strings to generate name variations of.
- **preprocess** (*list*) – List of preprocessing string methods to apply before generating variations.
- **swaps** (*list*) – List of tuples for str.replace(out, in).

Returns

Set of unique string variations.

Return type

set

pyrolite.util.text.**parse_entry**(*entry: str | None | float*, *regex='(\\s)*?(?P<value>[\\.\\w]+)(\\s)*?'*, *delimiter=','*, *values_only=True*, *first_only=True*, *replace_nan='None'*) → list[dict[str, str]] | dict[str, str] | list[str | float] | str | float

Parses an arbitrary string data entry to return values based on a regular expression containing named fields including 'value' (and any others). If the entry is of non-string type, this will return the value (e.g. int, float, NaN, None).

Parameters

- **entry** (*str*) – String entry which to search for the regex pattern.
- **regex** (*str*) – Regular expression to compile and use to search the entry for a value.
- **delimiter** (*str*) – Optional delimiter to split the string in case of multiple inclusion.
- **values_only** (*bool*) – Option to return only values (single or list), or to instead return the dictionary corresponding to the matches.
- **first_only** (*bool*) – Option to return only the first match, or else all matches

pyrolite.util.text.**split_records**(*data: str*, *delimiter: str = '\\r\\n'*) → list[str]

Splits records in a csv where quotation marks are used. Splits on a delimiter followed by an even number of quotation marks.

pyrolite.util.text.**slugify**(*value: str*, *delim: str = '-'*) → str

Normalizes a string, removes non-alpha characters, converts spaces to delimiters.

Parameters

- **value** (*str*) – String to slugify.
- **delim** (*str*) – Delimiter to replace whitespace with.

Return type
str

pyrolite.util.text.**int_to_alpha**(*num: int*) → str

Encode an integer into alpha characters, useful for sequences of axes/figures.

Parameters
int (*int*) – Integer to encode.

Returns
Alpha-encoding of a small integer.

Return type
str

pyrolite.util.web

pyrolite.util.web.**urlify**(*url: str*)

Strip a string to return a valid URL.

pyrolite.util.web.**have_internet_connection**(*target: str = 'pypi.org'*, *secure: bool = True*) → bool

Tests for an active internet connection, based on an optionally specified target.

Parameters
target (*str*) – URL to check connectivity, defaults to www.google.com

Returns
Boolean indication of whether a HTTP connection can be established at the given url.

Return type
bool

pyrolite.util.web.**download_file**(*url: str*, *encoding: str | None = 'UTF-8'*, *postprocess: Callable | None = None*) → str | bytes | None

Downloads a specific file from a url.

Parameters

- **url** (*str*) – URL of specific file to download.
- **encoding** (*str*) – String encoding.
- **postprocess** (*Callable*) – Callable function to post-process the requested content.

pyrolite.util.time

pyrolite.util.time.**listify**(*df: DataFrame*, *axis: int = 1*) → DataFrame

Consdense text information across columns into a single list.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe (or slice of dataframe) to condense along axis.
- **axis** (*int*) – Axis to condense along.

pyrolite.util.time.**age_name**(*agenamelist: list[str]*, *prefixes: list[str] | None = None*, *suffixes: list | None = None*) → str

Condenses an agename list to a specific agename, given a subset of ambiguous_names.

Parameters

- **agenamelist** (*list*) – List of name components (i.e. *[Eon, Era, Period, Epoch]*)

- **prefixes** (*list | None*) – Name components which occur prior to the higher order classification (e.g. *"Upper Triassic"*).
- **suffixes** (*list*) – Name components which occur after the higher order classification (e.g. *"Cambrian Series 2"*).

pyrolite.util.time.**import_colors**(*filename: Path | str = PosixPath('/home/docs/checkouts/pyrolite/checkouts/develop/pyrolite/data/timescale/timecolors.cs delim: str = '/'*) → dict[str, tuple[float, float, float, float]]

Import a list of timescale names with associated colors.

pyrolite.util.time.**timescale_reference_frame**(*filename: str | Path | None = None, info_cols: list[str] | None = None, color_info: dict[str, tuple[float, float, float, float] | str] | None = None*) → DataFrame

Rearrange the text-based timescale dataframe. Utility function for timescale class.

Parameters

- **filename** (*str | pathlib.Path*) – File from which to generate the timescale information.
- **info_cols** (*list*) – List of columns beyond hierarchial group labels (e.g. Eon, Era..).

Returns
Dataframe containing timescale information.

Return type
pandas.DataFrame

class pyrolite.util.time.**Timescale**(*filename: str | Path | None = None*)

build()

Build the timescale from data within file.

text2age(*entry: str, nulls: list[str | float | None] | None = None*) → tuple[float, float] | list[tuple[float, float]]

Converts a text-based age to the corresponding age range (in Ma).

String-based entries return (max_age, min_age). Collection-based entries return a list of tuples.

Parameters
entry (*str*) – String name, or series of string names, for geological age range.

Returns
Tuple or list of tuples.

Return type
tuple | list[tuple]

named_age(*age: float, level: str = 'Specific', **kwargs*) → str | None

Converts a numeric age (in Ma) to named age at a specific level.

Parameters

- **age** (*float*) – Numeric age in Ma.
- **level** (str, *{'Supereon', 'Eon', 'Era', 'Period', 'Superepoch', 'Epoch', 'Age', 'Specific'}*) – Level of specificity.

Returns
String representation for the entry.

Return type
str

pyrolite.util.math

pyrolite.util.math.**eigsorted**(*cov: ndarray[tuple[int, int], dtype[floating]]*)

Returns arrays of eigenvalues and eigenvectors sorted by magnitude.

Parameters
cov (*numpy.ndarray*) – Covariance matrix to extract eigenvalues and eigenvectors from.

Returns

- **vals** (*numpy.ndarray*) – Sorted eigenvalues.
- **vecs** (*numpy.ndarray*) – Sorted eigenvectors.

pyrolite.util.math.**augmented_covariance_matrix**(*M: ndarray[tuple[int, int], dtype[number]]*, *C: ndarray[tuple[int, int], dtype[floating]]*)

Constructs an augmented covariance matrix from means M and covariance matrix C.

Parameters

- **M** (*numpy.ndarray*) – Array of means.
- **C** (*numpy.ndarray*) – Covariance matrix.

Returns
Augmented covariance matrix A.

Return type
numpy.ndarray

Notes

Augmented covariance matrix constructed from mean of shape (D,) and covariance matrix of shape (D, D) as follows:

$$\begin{array}{c|c} -1 & M.T \\ \hline M & C \end{array}$$

pyrolite.util.math.**interpolate_line**(*x: ndarray[tuple[int]]*, *y: ndarray[tuple[int]]*, *n: int = 0*, *logy: bool = False*) → tuple[ndarray[tuple[int], dtype[floating]], ndarray[tuple[int], dtype[floating]]]

Add intermediate evenly spaced points interpolated between given x-y coordinates, assuming the x points are the same.

Parameters

- **x** (*numpy.ndarray*) – 1D array of x values.
- **y** (*numpy.ndarray*) – ND array of y values.

pyrolite.util.math.**grid_from_ranges**(*X: ndarray*, *bins: int | list[int] = 100*, ***kwargs*) → tuple[ndarray, ...]

Create a meshgrid based on the ranges along columns of array X.

Parameters

- **X** (*numpy.ndarray*) – Array of shape *(samples, dimensions)* to create a meshgrid from.
- **bins** (*int | tuple*) – Shape of the meshgrid. If an integer, provides a square mesh. If a tuple, values for each column are required.

Return type
numpy.ndarray

Notes

Can pass keyword arg indexing = {'xy', 'ij'}

pyrolite.util.math.**flattengrid**(*grid: tuple[ndarray, ...]*) → ndarray[tuple[int, int]]

Convert a collection of arrays to a concatenated array of flattened components. Useful for passing meshgrid values to a function which accepts argumnets of shape *(samples, dimensions)*.

Parameters
grid (*list*) – Collection of arrays (e.g. a meshgrid) to flatten and concatenate.

Return type
numpy.ndarray

pyrolite.util.math.**linspc_**(*_min: float*, *_max: float*, *step: float = 0.0*, *bins: int = 20*) → ndarray[tuple[int], dtype[floating]]

Linear spaced array, with optional step for grid margins.

Parameters

- **_min** (*float*) – Minimum value for spaced range.
- **_max** (*float*) – Maximum value for spaced range.
- **step** (*float, 0.0*) – Step for expanding at grid edges. Default of 0.0 results in no expansion.
- **bins** (*int*) – Number of bins to divide the range (adds one by default).

Returns
Linearly-spaced array.

Return type
numpy.ndarray

pyrolite.util.math.**logspc_**(*_min: float*, *_max: float*, *step: float = 1.0*, *bins: int = 20*) → ndarray[tuple[int], dtype[floating]]

Log spaced array, with optional step for grid margins.

Parameters

- **_min** (*float*) – Minimum value for spaced range.
- **_max** (*float*) – Maximum value for spaced range.
- **step** (*float, 1.0*) – Step for expanding at grid edges. Default of 1.0 results in no expansion.
- **bins** (*int*) – Number of bins to divide the range (adds one by default).

Returns
Log-spaced array.

Return type
numpy.ndarray

pyrolite.util.math.**logrng_**(*v: list[float] | ndarray*, *exp: float = 0.0*) → tuple[float, float]

Range of a sample, where values <0 are excluded.

Parameters

- **v** (*list; list-like*) – Array of values to obtain a range from.
- **exp** (*float, (0, 1)*) – Fractional expansion of the range.

Returns
Min, max tuple.

Return type
tuple

pyrolite.util.math.**linrng_**(*v: list[float] | ndarray*, *exp: float = 0.0*) → tuple[float, float]

Range of a sample, where values <0 are included.

Parameters

- **v** (*list; list-like*) – Array of values to obtain a range from.
- **exp** (*float, (0, 1)*) – Fractional expansion of the range.

Returns
Min, max tuple.

Return type
tuple

pyrolite.util.math.**isclose**(*a: float | ndarray*, *b: float | ndarray*) → bool | ndarray[tuple[int, ...], dtype[bool]]

Implementation of np.isclose with equal nan.

Parameters
a,b (*float | numpy.ndarray*) – Numbers or arrays to compare.

Return type
bool

pyrolite.util.math.**is_numeric**(*obj*) → bool

Check for numerical behaviour.

Parameters
obj – Object to check.

Return type
bool

pyrolite.util.math.**significant_figures**(*n: float | ndarray*, *unc: float | ndarray | None = None*, *max_sf: int = 20*, *rtol: float = 1e-20*) → int | ndarray

Iterative method to determine the number of significant digits for a given float, optionally providing an uncertainty.

Parameters

- **n** (*float*) – Number from which to ascertain the significance level.
- **unc** (float, *None*) – Uncertainty, which if provided is used to derive the number of significant digits.
- **max_sf** (*int*) – An upper limit to the number of significant digits suggested.
- **rtol** (*float*) – Relative tolerance to determine similarity of numbers, used in calculations.

Returns
Number of significant digits.

Return type
int

pyrolite.util.math.**signify_digit**(*n: float | ndarray*, *unc: float | ndarray | None = None*, *leeway: int = 0*, *low_filter: bool = True*)

Reformats numbers to contain only significant_digits. Uncertainty can be provided to digits with relevant precision.

Parameters

- **n** (*float*) – Number to reformat
- **unc** (float, *None*) – Absolute uncertainty on the number, optional.
- **leeway** (*int, 0*) – Manual override for significant figures. Positive values will force extra significant figures; negative values will remove significant figures.
- **low_filter** (bool, *True*) – Whether to return *np.nan* in place of values which are within precision equal to zero.

Returns
Reformatted number.

Return type
float

Notes

- Will not pad 0s at the end or before floats.

pyrolite.util.math.**most_precise**(*arr: ndarray*) → float | ndarray

Get the most precise element from an array.

Parameters
arr (*numpy.ndarray*) – Array to obtain the most precise element/subarray from.

Returns
Returns the most precise array element (for ndim=1), or most precise subarray (for ndim > 1).

Return type
float | numpy.ndarray

pyrolite.util.math.**equal_within_significance**(*arr: ndarray*, *equal_nan: bool = False*, *rtol: float = 1e-15*) → bool | ndarray

Test whether elements of an array are equal within the precision of the least precise.

Parameters

- **arr** (*numpy.ndarray*) – Array to test.
- **equal_nan** (bool, *False*) – Whether to consider *np.nan* elements equal to one another.
- **rtol** (*float*) – Relative tolerance for comparison.

Return type
bool | numpy.ndarray(bool)

pyrolite.util.math.**helmert_basis**(*D: int*, *full: bool = False*, ***kwargs*) → ndarray[tuple[int, int]]

Generate a set of orthogonal basis vectors in the form of a helmert matrix.

Parameters
D (*int*) – Dimension of compositional vectors.

Returns
(D-1, D) helmert matrix corresponding to default orthogonal basis.

Return type
numpy.ndarray

pyrolite.util.math.**symbolic_helmert_basis**(*D: int*, *full: bool = False*) → DenseMatrix

Get a symbolic representation of a Helmert Matrix.

Parameters

- **D** (*int*) – Order of the matrix. Equivalent to dimensionality for compositional data analysis.
- **full** (*bool*) – Whether to return the full matrix, or alternatively exclude the first row. Analogous to the option for *scipy.linalg.helmert*.

Return type
sympy.matrices.dense.DenseMatrix

pyrolite.util.math.**on_finite**(*X: ndarray*, *f: Callable*) → ndarray

Calls a function on an array ignoring np.nan and +/- np.inf. Note that the shape of the output may be different to that of the input.

Parameters

- **X** (*numpy.ndarray*) – Array on which to perform the function.
- **f** (*Callable*) – Function to call on the array.

Return type
numpy.ndarray

pyrolite.util.math.**nancov**(*X: ndarray[tuple[int, int]]*) → ndarray[tuple[int, int]]

Generates a covariance matrix excluding nan-components.

Parameters
X (*numpy.ndarray*) – Input array for which to derive a covariance matrix.

Return type
numpy.ndarray

pyrolite.util.math.**solve_ratios**(**eqs*, *evaluate: bool = True*) → list[float]

Solve a ternary system (top-left-right) given two constraints on two ratios, which together describe intersecting lines/a point.

pyrolite.util.lambdas

pyrolite.util.lambdas.**calc_lambdas**(*df: DataFrame*, *params: list | str | None = None*, *degree: int = 4*, *exclude: list[str] | None = None*, *algorithm: str = 'ONeill'*, *anomalies: list[str] | None = None*, *fit_tetrads: bool = False*, *sigmas: float | list[float] | ndarray | None = None*, *add_uncertainties: bool = False*, *add_X2: bool = False*, ***kwargs*) → DataFrame

Parameterises values based on linear combination of orthogonal polynomials over a given set of values for independent variable *x* **[#pyrolite.util.lambdas.calc_lambdas_1]_** . This function expects to recieve data already normalised and log-transformed.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe containing REE Data.
- **params** (*list | str*) – Pre-computed parameters for the orthogonal polynomials (a list of tuples). Optionally specified, otherwise defaults the parameterisation as in O'Neill (2016). **[#pyrolite.util.lambdas.calc_lambdas_1]_** If a string is supplied, *"O'Neill (2016)"* or similar will give the original defaults, while *"full"* will use all of the REE (including Eu) as a basis for the orthogonal polynomials.
- **degree** (*int*) – Degree of orthogonal polynomial fit.
- **exclude** (*list*) – REE to exclude from the *fit*.
- **algorithm** (*str*) – Algorithm to use for fitting the orthogonal polynomials.
- **anomalies** (*list*) – List of relative anomalies to append to the dataframe.

- **fit_tetrads** (*bool*) – Whether to fit tetrad functions in addition to orthogonal polynomial functions. This will force the use of the optimization algorithm.
- **sigmas** (*float | numpy.ndarray*) – Single value or 1D array of observed value uncertainties.
- **add_uncertainties** (*bool*) – Whether to append estimated parameter uncertainties to the dataframe.
- **add_X2** (*bool*) – Whether to append the chi-squared values (2) to the dataframe.

Return type
pandas.DataFrame

See also

orthogonal_polynomial_constants(), *lambda_lnREE()*, *normalize_to()*

References

pyrolite.util.lambdas.params

Functions to generate parameters for the construction of orthogonal polynomials which are used to fit REE patterns.

pyrolite.util.lambdas.params.**orthogonal_polynomial_constants**(*xs: ndarray*, *degree: int = 3*, *rounding: int | None = None*, *tol: float = 1e-14*) → list[tuple[float, ...]]

Finds the parameters $(\beta_0), (\gamma_0, \gamma_1), (\delta_0, \delta_1, \delta_2)$ etc. for constructing orthogonal polynomial functions *f(x)* over a fixed set of values of independent variable *x*. Used for obtaining lambda values for dimensional reduction of REE data **[#pyrolite.util.lambdas.params.orthogonal_polynomial_constants_1]_**.

Parameters

- **xs** (*numpy.ndarray*) – Indexes over which to generate the orthogonal polynomials.
- **degree** (*int*) – Maximum polynomial degree. E.g. 2 will generate constant, linear, and quadratic polynomial components.
- **rounding** (*int*) – Precision for the orthogonal polynomial coefficents.
- **tol** (*float*) – Convergence tolerance for solver.

Returns
List of tuples corresponding to coefficients for each of the polynomial components. I.e the first tuple will be empty, the second will contain a single coefficient etc.

Return type
list

Notes

Parameters are used to construct orthogonal polymomials of the general form:

$$
\begin{aligned}
f(x) = a_0 \\
&+ a_1 * (x - \beta) \\
&+ a_2 * (x - \gamma_0) * (x - \gamma_1) \\
&+ a_3 * (x - \delta_0) * (x - \delta_1) * (x - \delta_2)
\end{aligned}
$$

See also

calc_lambdas(), *lambda_lnREE()*

References

pyrolite.util.lambdas.params.**parse_sigmas**(*size: int*, *sigmas: float | ndarray | None = None*) → ndarray

Disambigaute a value or set of sigmas for a dataset for use in lambda-fitting algorithms.

Parameters

sigmas (*float | numpy.ndarray*) – 2D array of REE uncertainties. Values as fractional uncertaintes (i.e. σ_{REE}/REE).

Returns

sigmas – 1D array of sigmas (σ_{REE}/REE).

Return type

float | numpy.ndarray

Notes

Note that the y-array is passed here only to infer the shape which should be assumed by the uncertainty array. Through propagation of uncertainties, the uncertainty on the natural logarithm of the normalised REE values are equivalent to σ_{REE}/REE where the uncertainty in the reference composition is assumed to be zero. Thus, standard deviations of 1% in REE will result in $\sigma = 0.01$ for the log-transformed REE. If no sigmas are provided, 1% uncertainty will be assumed and an array of 0.01 will be returned.

>

pyrolite.util.lambdas.eval

Generation and evalutation of orthogonal polynomial and tetrad functions from sets of parameters (the sequence of polymomial roots and tetrad centres and widths).

pyrolite.util.lambdas.eval.**lambda_poly**(*x: ndarray*, *ps: tuple[float]*) → ndarray

Evaluate polynomial *lambda_n(x)* given a tuple of parameters *ps* with length equal to the polynomial degree.

Parameters

- **x** (*numpy.ndarray*) – X values to calculate the function at.
- **ps** (*tuple*) – Parameter set tuple. E.g. parameters *(a, b)* from $f(x) = (x - a)(x - b)$.

Return type

numpy.ndarray

pyrolite.util.lambdas.eval.**tetrad**(*x: ndarray*, *centre: float*, *width: float*) → ndarray

Evaluate $f(z)$ describing a tetrad with specified centre and width.

Parameters

- **x** (*numpy.ndarray*) – Radii.
- **centre** (*float*) – Centre of the tetrad.
- **width** (*float*) – Width of the tetrad.

Return type

numpy.ndarray

pyrolite.util.lambdas.eval.**get_tetrads_function**(*params: list[tuple[float, ...]] | None = None*) → Callable

pyrolite.util.lambdas.eval.**get_lambda_poly_function**(*lambdas: ndarray*, *params: list[tuple[float, ...]] | None = None*, *radii: ndarray | None = None*, *degree: int = 5*) → Callable

Expansion of lambda parameters back to the original space. Returns a function which evaluates the sum of the orthogonal polynomials at given *x* values.

Parameters

- **lambdas** (*numpy.ndarray*) – Lambda values to weight combination of polynomials.
- **params** (*list[tuple]*) – Parameters for the orthogonal polynomial decomposition.
- **radii** (*numpy.ndarray*) – Radii values used to construct the lambda values. **[#note_1]_**
- **degree** (*int*) – Degree of the orthogonal polynomial decomposition. **[#note_1]_**

See also

calc_lambdas(), *orthogonal_polynomial_constants()*, *lambda_lnREE()*

Notes

pyrolite.util.lambdas.eval.**get_function_components**(*radii: ndarray*, *params: list[tuple[float, ...]] | None = None*, *fit_tetrads: bool = False*, *tetrad_params: list[tuple[float, ...]] | None = None*, *degree: int = 5*, ***kwargs*) → tuple[list[str], list[float], list[ndarray]]

>

pyrolite.util.lambdas.oneill

Linear algebra methods for fitting a series of orthogonal polynomial functions to REE patterns.

pyrolite.util.lambdas.oneill.**get_polynomial_matrix**(*radii: ndarray | list[float]*, *params: list[tuple[float, ...]] | None = None*) → ndarray

Create the matrix *A* with polynomial components across the columns, and increasing order down the rows.

Parameters

- **radii** (*list, numpy.ndarray*) – Radii at which to evaluate the orthogonal polynomial.
- **params** (*tuple*) – Tuple of constants for the orthogonal polynomial.

Return type
numpy.ndarray

See also

orthogonal_polynomial_constants()

pyrolite.util.lambdas.oneill.**lambdas_ONeill2016**(*df: DataFrame*, *radii: ndarray*, *params: list[tuple[float, ...]] | None = None*, *sigmas: float | ndarray | None = None*, *add_X2: bool = False*, *add_uncertainties: bool = False*, ***kwargs*)

Implementation of the original algorithm. [#pyrolite.util.lambdas.oneill.lambdas_ONeill2016_1]_

Parameters

- **df** (*pandas.DataFrame | pandas.Series*) – Dataframe of REE data, with sample analyses organised by row.
- **radii** (*list, numpy.ndarray*) – Radii at which to evaluate the orthogonal polynomial.
- **params** (*tuple*) – Tuple of constants for the orthogonal polynomial.
- **sigmas** (*float | numpy.ndarray*) – Single value or 1D array of normalised observed value uncertainties (σ_{REE}/REE).
- **add_X2** (*bool*) – Whether to append the chi-squared values (2) to the dataframe/series.
- **add_uncertainties** (*bool*) – Append parameter standard errors to the dataframe/series.

Return type
pandas.DataFrame

See also

orthogonal_polynomial_constants(), *lambda_lnREE()*

References

>

pyrolite.util.lambdas.opt

Functions for optimization-based REE profile fitting and parameter uncertainty estimation.

pyrolite.util.lambdas.opt.**pcov_from_jac**(*jac: ndarray*) → ndarray

Extract a covariance matrix from a Jacobian matrix returned from `scipy.optimize` functions.

Parameters
jac (*numpy.ndarray*) – Jacobian array.

Returns
pcov – Square covariance array; this hasn't yet been scaled by residuals.

Return type
numpy.ndarray

pyrolite.util.lambdas.opt.**linear_fit_components**(*y: ndarray*, *x0: ndarray | list[float]*, *func_components: list[ndarray]*, *sigmas: float | ndarray | None = None*) → tuple[ndarray, ndarray, ndarray]

Fit a weighted sum of function components using linear algebra.

Parameters

- **y** (*numpy.ndarrayndarray*) – Array of target values to fit.
- **x0** (*numpy.ndarray*) – Starting guess for the function weights.
- **func_components** (*list[numpy.ndarray]*) – List of arrays representing static/evaluated function components.

- **sigmas** (*float | numpy.ndarray*) – Single value or 1D array of normalised observed value uncertainties (σ_{REE}/REE).

Returns
B, s, 2 – Arrays for the optimized parameter values (B; (n, d)), parameter uncertaintes (s, 1; (n, d)) and chi-chi_squared (2; (n, 1)).

Return type
numpy.ndarray

pyrolite.util.lambdas.opt.**optimize_fit_components**(*y: ~numpy.ndarray, x0: ~numpy.ndarray | list[float], func_components: list[~numpy.ndarray], residuals_function: ~collections.abc.Callable = <function _residuals_func>, sigmas: float | ~numpy.ndarray | None = None*) → tuple[ndarray, ndarray, ndarray]

Fit a weighted sum of function components using `scipy.optimize.least_squares()`.

Parameters

- **y** (*numpy.ndarray*) – Array of target values to fit.
- **x0** (*numpy.ndarray*) – Starting guess for the function weights.
- **func_components** (*list (numpy.ndarray)*) – List of arrays representing static/evaluated function components.
- **redsiduals_function** (*callable*) – Callable funciton to compute residuals which accepts ordered arguments for weights, target values and function components.
- **sigmas** (*float | numpy.ndarray*) – Single value or 1D array of normalised observed value uncertainties (σ_{REE}/REE).

Returns
B, s, 2 – Arrays for the optimized parameter values (B; (n, d)), parameter uncertaintes (s, 1; (n, d)) and chi-chi_squared (2; (n, 1)).

Return type
numpy.ndarray

pyrolite.util.lambdas.opt.**lambdas_optimize**(*df: DataFrame | Series, radii: ndarray, params: list[tuple[float, ...]] | None = None, fit_tetrads: bool = False, tetrad_params: list[tuple[float, ...]] | None = None, fit_method: str = 'opt', sigmas: float | ndarray | None = None, add_uncertainties: bool = False, add_X2: bool = False, **kwargs*) → DataFrame | Series

Parameterises values based on linear combination of orthogonal polynomials over a given set of values for independent variable *x*. **[#pyrolite.util.lambdas.opt.lambdas_optimize_1]_**

Parameters

- **df** (*pandas.DataFrame | pandas.Series*) – Target data to fit. For geochemical data, this is typically normalised so we can fit a smooth function.
- **radii** (*list, numpy.ndarray*) – Radii at which to evaluate the orthogonal polynomial.
- **params** (*list*) – Orthogonal polynomial coefficients (see *orthogonal_polynomial_constants*).
- **fit_tetrads** (*bool*) – Whether to also fit the patterns for tetrads.
- **tetrad_params** (*list*) – List of parameter sets for tetrad functions.
- **fit_method** (*str*) – Which fit method to use: *"optimization"* or *"linear"*.

- **sigmas** (*float | numpy.ndarray*) – Single value or 1D array of observed value uncertainties.
- **add_uncertainties** (*bool*) – Whether to append estimated parameter uncertainties to the dataframe/series.
- **add_X2** (*bool*) – Whether to append the chi-squared values (2) to the dataframe/series.

Returns
Optimial results for weights of orthogonal polymomial regression (*lambdas*).

Return type
pandas.DataFrame | pandas.Series

See also

orthogonal_polynomial_constants(), None

References

>

pyrolite.util.lambdas.plot

Functions for the visualisation of reconstructed and deconstructed parameterised REE profiles based on parameterisations using 'lambdas' (and tetrad-equivalent weights 'taus').

pyrolite.util.lambdas.plot.**plot_lambdas_components**(*lambdas: Series | ndarray*, *params: list[tuple[float, ...]] | None = None*, *ax: Axes | None = None*, ***kwargs*) → Axes

Plot a decomposed orthogonal polynomial from a single set of lambda coefficients.

Parameters

- **lambdas** – 1D array of lambdas.
- **params** (*list*) – List of orthongonal polynomial parameters, if defaults are not used.
- **ax** (*matplotlib.axes.Axes*) – Optionally specified axes to plot on.
- **index** (*str*) – Index to use for the plot (one of *"index", "radii", "z"*).

Return type
matplotlib.axes.Axes

pyrolite.util.lambdas.plot.**plot_tetrads_components**(*taus: Series | ndarray*, *tetrad_params: list[tuple[float, ...]] | None = None*, *ax: Axes | None = None*, *index: str = 'radii'*, *logy: bool = True*, *drop0: bool = True*, ***kwargs*) → Axes

Individually plot the four tetrad components for one set of $ au$s.

Parameters

- **taus** (*numpy.ndarray*) – 1D array of $ au$ tetrad function coefficients.
- **tetrad_params** (*list*) – List of tetrad parameters, if defaults are not used.
- **ax** (*matplotlib.axes.Axes*) – Optionally specified axes to plot on.
- **index** (*str*) – Index to use for the plot (one of *"index", "radii", "z"*).
- **logy** (*bool*) – Whether to log-scale the y-axis.

- **drop0** (*bool*) – Whether to remove zeroes from the outputs such that individual tetrad functions are shown only within their respective bounds (and not across the entire REE, where their effective values are zero).

pyrolite.util.lambdas.plot.**plot_profiles**(*coefficients: Series | ndarray*, *tetrads: bool = False*, *params: list[tuple[float, ...]] | None = None*, *tetrad_params: list[tuple[float, ...]] | None = None*, *ax: Axes | None = None*, *index: str = 'radii'*, *logy: bool = False*, ***kwargs*) → Axes

Plot the reconstructed REE profiles of a 2D dataset of coefficients ($lambda$s, and optionally taus).

Parameters

- **coefficients** (*numpy.ndarray*) – 2D array of $lambda$ orthogonal polynomial coefficients, and optionally including tau tetrad function coefficients in the last four columns (where *tetrads=True*).
- **tetrads** (*bool*) – Whether the coefficient array contains tetrad coefficients (taus).
- **params** (*list*) – List of orthongonal polynomial parameters, if defaults are not used.
- **tetrad_params** (*list*) – List of tetrad parameters, if defaults are not used.
- **ax** (*matplotlib.axes.Axes*) – Optionally specified axes to plot on.
- **index** (*str*) – Index to use for the plot (one of *"index", "radii", "z"*).
- **logy** (*bool*) – Whether to log-scale the y-axis.

Return type
matplotlib.axes.Axes

pyrolite.util.lambdas.transform

Functions for transforming ionic radii to and from atomic number for the visualisation of REE patterns.

pyrolite.util.lambdas.transform.**REE_z_to_radii**(*z: float | list[float] | ndarray*, *fit: Callable | None = None*, *degree: int = 7*, ***kwargs*) → float | ndarray

Estimate the ionic radii which would be approximated by a given atomic number based on a provided (or calculated) fit for the Rare Earth Elements.

Parameters

- **z** (*float | list | numpy.ndarray*) – Atomic nubmers to be converted.
- **fit** (*callable*) – Callable function optionally specified; if not specified it will be calculated from Shannon Radii.
- **degree** (*int*) – Degree of the polynomial fit between atomic number and radii.

Returns
r – Approximate atomic nubmers for given radii.

Return type
float | numpy.ndarray

pyrolite.util.lambdas.transform.**REE_radii_to_z**(*r: float | list[float] | ndarray*, *fit: Callable | None = None*, *degree: int = 7*, ***kwargs*) → float | ndarray

Estimate the atomic number which would be approximated by a given ionic radii based on a provided (or calculated) fit for the Rare Earth Elements.

Parameters

- **r** (*float | list | numpy.ndarray*) – Radii to be converted.

- **fit** (*callable*) – Callable function optionally specified; if not specified it will be calculated from Shannon Radii.
- **degree** (*int*) – Degree of the polynomial fit between radii and atomic number.

Returns
z – Approximate atomic numbers for given radii.

Return type
float | numpy.ndarray

pyrolite.util.distributions

pyrolite.util.distributions.**get_scaler**(**fs: Callable*) → Callable

Generate a function which will transform columns of an array based on input functions (e.g. *np.log* will log-transform the x values, *None, np.log* will log-transform the y values but not the x).

Parameters
fs – A series of functions to apply to subsequent axes of an array.

pyrolite.util.distributions.**sample_kde**(*data: ~numpy.ndarray[tuple[int, int]], samples: ~numpy.ndarray[tuple[int, int]], renorm: bool = False, transform: ~collections.abc.Callable = <function <lambda>>, bw_method: str | float | ~collections.abc.Callable | None = None*) → ndarray[tuple[int, int]]

Sample a Kernel Density Estimate at points or a grid defined.

Parameters

- **data** (*numpy.ndarray*) – Source data to estimate the kernel density estimate; observations should be in rows (*npoints, ndim*).
- **samples** (*numpy.ndarray*) – Coordinates to sample the KDE estimate at (*npoints, ndim*).
- **transform** – Transformation used prior to kernel density estimate.
- **bw_method** (*str, float, callable*) – Method used to calculate the estimator bandwidth. See *scipy.stats.gaussian_kde*.

Return type
numpy.ndarray

pyrolite.util.distributions.**sample_ternary_kde**(*data: ~numpy.ndarray[tuple[int, int]], samples: ~numpy.ndarray[tuple[int, int]], transform: ~collections.abc.Callable = <function ILR>*) → ndarray[tuple[int, int]]

Sample a Kernel Density Estimate in ternary space points or a grid defined by samples.

Parameters

- **data** (*numpy.ndarray*) – Source data to estimate the kernel density estimate (*npoints, ndim*).
- **samples** (*numpy.ndarray*) – Coordinates to sample the KDE estimate at (*npoints, ndim*)..
- **transform** – Log-transformation used prior to kernel density estimate.

Return type
numpy.ndarray

pyrolite.util.distributions.**lognorm_to_norm**(*mu: float | ndarray, s: float | ndarray*) → tuple[float | ndarray, float | ndarray]

Calculate mean and variance for a normal random variable from the lognormal parameters *mu* and *s*.

Parameters

- **mu** (*float*) – Parameter *mu* for the lognormal distribution.
- **s** (*float*) – *sigma* for the lognormal distribution.

Returns

- **mean** (*float*) – Mean of the normal distribution.
- **sigma** (*float*) – Variance of the normal distribution.

pyrolite.util.distributions.**norm_to_lognorm**(*mean: float | ndarray*, *sigma: float | ndarray*, *exp: bool = True*) → tuple[float | ndarray, float | ndarray]

Calculate *mu* and *sigma* parameters for a lognormal random variable with a given mean and variance. Lognormal with parameters *mean* and *sigma*.

Parameters

- **mean** (*float*) – Mean of the normal distribution.
- **sigma** (*float*) – *sigma* of the normal distribution.
- **exp** (*bool*) – If using the `scipy.stats` parameterisation; this uses *scale = np.exp(mu)*.

Returns

- **mu** (*float*) – Parameter *mu* for the lognormal distribution.
- **s** (*float*) – *sigma* of the lognormal distribution.

pyrolite.util.synthetic

Utility functions for creating synthetic (geochemical) data.

pyrolite.util.synthetic.**random_cov_matrix**(*dim: int*, *sigmas: list[float] | ndarray[tuple[int], dtype[floating]] | None = None*, *validate: bool = False*, *seed: int | None = None*) → ndarray[tuple[int, int]]

Generate a random covariance matrix which is symmetric positive-semidefinite.

Parameters

- **dim** (*int*) – Dimensionality of the covariance matrix.
- **sigmas** (*numpy.ndarray*) – Optionally specified sigmas for the variables, 1D.
- **validate** (*bool*) – Whether to validate output.

Returns
Covariance matrix of shape *(dim, dim)*.

Return type
numpy.ndarray

Todo

- Implement a characteristic scale for the covariance matrix.

pyrolite.util.synthetic.**random_composition**(*size: int = 1000*, *D: int = 4*, *mean: list | ndarray[tuple[int], dtype[floating]] | None = None*, *cov: ndarray[tuple[int, int], dtype[floating]] | None = None*, *propnan: float = 0.1*, *missing_columns: int | tuple | None = None*, *missing: str | None = None*, *seed: int | None = None*) → ndarray[tuple[int, int], dtype[floating]]

Generate a simulated random unimodal compositional dataset, optionally with missing data.

Parameters

- **size** (*int*) – Size of the dataset.
- **D** (*int*) – Dimensionality of the dataset.
- **mean** (*numpy.ndarray*) – Optional specification of mean composition.
- **cov** (*numpy.ndarray*) – Optional specification of covariance matrix (in log space).
- **propnan** (*float*) – Proportion of missing values in the output dataset.
- **missing_columns** (*int | tuple*) – Specification of columns to be missing. If an integer is specified, interpreted to be the number of columns containin missing data (at a proportion defined by *propnan*). If a tuple or list, the specific columns to contain missing data.
- **missing** (*str*) – Missingness pattern. If not *None*, one of *"MCAR", "MAR", "MNAR"*.
  - If *missing* = *"MCAR"*, data will be missing at random.
  - If *missing* = *"MAR"*, data will be missing with some relationship to other parameters.
  - If *missing* = *"MNAR"*, data will be thresholded at some lower bound.
- **seed** (*int*) – Random seed to use, optionally specified.

Returns
Simulated dataset with missing values.

Return type
numpy.ndarray

Todo

- Add feature to translate rough covariance in D to logcovariance in D-1
- Update the `missing = "MAR"` example to be more realistic/variable.

pyrolite.util.synthetic.**normal_frame**(*columns: list | None = None*, *size: int = 10*, *mean: list | ndarray[tuple[int], dtype[floating]] | None = None*, ***kwargs*) → DataFrame

Creates a pandas.DataFrame with samples from a single multivariate-normal distributed composition.

Parameters

- **columns** (*list*) – List of columns to use for the dataframe. These won't have any direct impact on the data returned, and are only for labelling.
- **size** (*int*) – Index length for the dataframe.
- **mean** (*numpy.ndarray | list*) – Optional specification of mean composition.

Return type
pandas.DataFrame

Notes

See also: *random_composition()*.

pyrolite.util.synthetic.**normal_series**(*index: list | Index | None = None*, *mean: list | ndarray[tuple[int], dtype[floating]] | None = None*, ***kwargs*) → Series

Creates a pandas.Series with a single sample from a single multivariate-normal distributed composition.

Parameters

- **index** (*list*) – List of indexes for the series. These won't have any direct impact on the data returned, and are only for labelling.
- **mean** (numpy.ndarray, *None*) – Optional specification of mean composition.

Return type
pandas.Series

Notes

See also: *random_composition()*.

pyrolite.util.synthetic.**example_spider_data**(*start: str = 'EMORB_SM89'*, *norm_to: str | None = 'PM_PON'*, *size: int = 120*, *noise_level: float = 0.5*, *offsets: dict[str, float] | None = None*, *units: str = 'ppm'*) → DataFrame

Generate some random data for demonstrating spider plots.

By default, this generates a composition based around EMORB, normalised to Primitive Mantle.

Parameters

- **start** (*str*) – Composition to start with.
- **norm_to** (*str*) – Composition to normalise to. Can optionally specify *None*.
- **size** (*int*) – Number of observations to include (index length).
- **noise_level** (*float*) – Log-units of noise (1sigma).
- **offsets** (*dict*) – Dictionary of offsets in log-units (in log units).
- **units** (*str*) – Units to use before conversion. Should have no effect other than reducing calculation times if *norm_to* is *None*.

Returns
df – Dataframe of example synthetic data.

Return type
pandas.DataFrame

pyrolite.util.synthetic.**example_patterns_from_parameters**(*fit_parameters: ndarray*, *radii: ndarray | None = None*, *n: int = 100*, *proportional_noise: float = 0.15*, *includes_tetrads: bool = False*, *columns: list[str] | None = None*) → DataFrame

pyrolite.util.spatial

Baisc spatial utility functions.

pyrolite.util.spatial.**great_circle_distance**(*a: ndarray[tuple[int]] | ndarray[tuple[int, int]], b: ndarray[tuple[int]] | ndarray[tuple[int, int]] | None = None, absolute: bool = False, degrees: bool = True, r: float = 6371.0088, method: str | None = None, dtype: str | dtype = 'float32', max_memory_fraction: float = 0.25*) → ndarray[tuple[int], dtype[floating]]

Calculate the great circle distance between two lat, long points.

Parameters

- **a, b** (*numpy.ndarray*) – Lat-Long points or arrays to calculate distance between. If only one array is specified, a full distance matrix (i.e. calculate a point-to-point distance for every combination of points) will be returned.
- **absolute** (*bool*) – Whether to return estimates of on-sphere distances [True], or simply return the central angle between the points.
- **degrees** (*bool*) – Whether lat-long coordinates are in degrees [True] or radians [False].
- **r** (*float*) – Earth radii for estimating absolute distances.
- **method** (*str*) – Which method to use for great circle distance calculation. Defaults to the Vicenty formula.
- **dtype** (*numpy.dtype*) – Data type for distance arrays, to constrain memory management.
- **max_memory_fraction** (*float*) – Constraint to switch to calculating mean distances where *matrix=True* and the distance matrix requires greater than a specified fraction of total avaialbe physical memory.

pyrolite.util.spatial.**piecewise**(*segment_ranges: list[tuple[float, float]], segments: int = 2, output_fmt: ~numpy.dtype | ~collections.abc.Callable = <class 'numpy.float64'>*) → Generator[ndarray, None, None]

Generator to provide values of quantizable paramaters which define a grid, here used to split up queries from databases to reduce load.

Parameters

- **segment_ranges** (*list*) – List of segment ranges to create a grid from.
- **segments** (*int*) – Number of segments.
- **output_fmt** – Function to call on the output.

pyrolite.util.spatial.**spatiotemporal_split**(*segments: int = 4, nan_lims: tuple[float, float] | None = None, **kwargs*) → Generator[dict, None, None]

Creates spatiotemporal grid using piecewise function and arbitrary ranges for individial kw-parameters (e.g. age=(0., 450.)), and sequentially returns individial grid cell attributes.

Parameters

- **segments** (*int*) – Number of segments.
- **nan_lims** (*tuple[float,float]*) – Specificaiton of NaN indexes for missing boundaries.

Yields

dict – Iteration through parameter sets for each cell of the grid.

pyrolite.util.spatial.**NSEW_2_bounds**(*cardinal: dict, order: list | None = None*) → list

Translates cardinal points to xy points in the form of bounds. Useful for converting to the format required for WFS from REST style queries.

Parameters

- **cardinal** (*dict*) – Cardinally-indexed point bounds.
- **order** (*list*) – List indicating order of returned x-y bound coordinates.

Returns
x-y indexed extent values in the specified order.

Return type
list

pyrolite.util.spatial.**levenshtein_distance**(*seq_one: str | list[str]*, *seq_two: str | list[str]*) → int

Compute the Levenshtein Distance between two sequences with comparable items. Adapted from Wiki pseudocode.

Parameters
seq_one, seq_two (*str | list[str]*) – Sequences to compare.

Return type
int

pyrolite.util.resampling

Utilities for (weighted) bootstrap resampling applied to geoscientific point-data.

pyrolite.util.resampling.**univariate_distance_matrix**(*a*, *b=None*, *distance_metric=None*)

Get a distance matrix for a single column or array of values (here used for ages).

Parameters

- **a, b** (*numpy.ndarray*) – Points or arrays to calculate distance between. If only one array is specified, a full distance matrix (i.e. calculate a point-to-point distance for every combination of points) will be returned.
- **distance_metric** – Callable function f(a, b) from which to derive a distance metric.

Returns
2D distance matrix.

Return type
numpy.ndarray

pyrolite.util.resampling.**get_spatiotemporal_resampling_weights**(*df*, *spatial_norm=1.8*, *temporal_norm=38*, *latlong_names=None*, *age_name='Age'*, *max_memory_fraction=0.25*, *normalized_weights=True*, ***kwargs*)

Takes a dataframe with lat, long and age and returns a sampling weight for each sample which is essentailly the inverse of the mean distance to other samples.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to calculate weights for.
- **spatial_norm** (*float*) – Normalising constant for spatial measures (1.8 arc degrees).
- **temporal_norm** (*float*) – Normalising constant for temporal measures (38 Mya).
- **latlong_names** (*list*) – List of column names referring to latitude and longitude.
- **age_name** (*str*) – Column name corresponding to geological age or time.

- **max_memory_fraction** (*float*) – Constraint to switch to calculating mean distances where *matrix=True* and the distance matrix requires greater than a specified fraction of total avaialbe physical memory. This is passed on to *~pyrolite.util.spatial.great_circle_distance*.
- **normalized_weights** (*bool*) – Whether to renormalise weights to unity.

Returns
weights – Sampling weights.

Return type
numpy.ndarray

Notes

This function is equivalent to Eq(1) from Keller and Schone:

$$W_i \propto 1 \Big/ \sum_{j=1}^{n} \Big(\frac{1}{((z_i - z_j)/a)^2 + 1} + \frac{1}{((t_i - t_j)/b)^2 + 1} \Big)$$

pyrolite.util.resampling.**add_age_noise**(*df*, *min_sigma=50*, *noise_level=1.0*, *age_name='Age'*, *age_uncertainty_name='AgeUncertainty'*, *min_age_name='MinAge'*, *max_age_name='MaxAge'*)

Add gaussian noise to a series of geological ages based on specified uncertainties or age ranges.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe with age data within which to look up the age name and add noise.
- **min_sigma** (*float*) – Minimum uncertainty to be considered for adding age noise.
- **noise_level** (*float*) – Scaling of the noise added to the ages. By default the uncertaines are unscaled, but where age uncertaines are specified and are the one standard deviation level this can be used to expand the range of noise added (e.g. to 2SD).
- **age_name** (*str*) – Column name for absolute ages.
- **age_uncertainty_name** (*str*) – Name of the column specifiying absolute age uncertainties.
- **min_age_name** (*str*) – Name of the column specifying minimum absolute ages (used where uncertainties are otherwise unspecified).
- **max_age_name** (*str*) – Name of the column specifying maximum absolute ages (used where uncertainties are otherwise unspecified).

Returns
df – Dataframe with noise-modified ages.

Return type
pandas.DataFrame

Notes

This modifies the dataframe which is input - be aware of this if using outside of the bootstrap resampling for which this was designed.

pyrolite.util.resampling.**spatiotemporal_bootstrap_resample**(*df: DataFrame*, *columns: list[str] | None = None*, *uncert: float | ndarray | Series | DataFrame | None = None*, *weights: ndarray | Series | None = None*, *niter: int = 100*, *categories: list | ndarray | Series | None = None*, *transform: Callable | None = None*, *bootstrap_method: str = 'smooth'*, *add_gaussian_age_noise: bool = True*, *metrics: tuple[Callable | str] | None = None*, *default_uncertainty: float = 0.02*, *relative_uncertainties: bool = True*, *noise_level: float = 1*, *age_name: str = 'Age'*, *latlong_names: tuple[str, str] | None = None*, ***kwargs*) → dict[str, DataFrame]

Resample and aggregate metrics from a dataframe, optionally aggregating by a given set of categories. Formulated specifically for dealing with resampling to address uneven sampling density in space and particularly geological time.

Parameters

- **df** (*pandas.DataFrame*) – Dataframe to resample.
- **columns** (*list*) – Columns to provide bootstrap resampled estimates for.
- **uncert** (*float | numpy.ndarray | pandas.Series | pandas.DataFrame*) – Fractional uncertainties for the dataset.
- **weights** (*numpy.ndarray | pandas.Series*) – Array of weights for resampling, if precomputed.
- **niter** (*int*) – Number of resampling iterations. This will be the minimum index size of the output metric dataframes.
- **categories** (*list | numpy.ndarray | pandas.Series*) – List of sample categories to group the ouputs by, which has the same size as the dataframe index.
- **transform** – Callable function to transform input data prior to aggregation functions. Note that the outputs will need to be inverse-transformed.
- **bootstrap_method** (*str*) – Which method to use to add gaussian noise to the input dataset parameters.
- **add_gaussian_age_noise** (*bool*) – Whether to add gassian noise to the input dataset ages, where present.
- **metrics** (*list*) – List of metrics to use for dataframe aggregation.
- **default_uncertainty** (*float*) – Default (fractional) uncertainty where uncertainties are not given.
- **relative_uncertainties** (*bool*) – Whether uncertainties are relative (*True*, i.e. fractional proportions of parameter values), or absolute (*False*)
- **noise_level** (*float*) – Multiplier for the random gaussian noise added to the dataset and ages.
- **age_name** (*str*) – Column name for geological age.
- **latlong_names** (*list*) – Column names for latitude and longitude, or equvalent orthogonal spherical spatial measures.

Returns
Dictionary of aggregated Dataframe(s) indexed by statistical metrics. If categories are specified, the dataframe(s) will have a hierarchical index of *categories, iteration*.

Return type
dict

pyrolite.util.missing

pyrolite.util.missing.**md_pattern**(*Y: ndarray[tuple[int, int]]*) → tuple[ndarray[tuple[int]], dict[int, dict[str, ndarray | int]]]

Get the missing data patterns from an array.

Parameters
Y (*numpy.ndarray | pandas.DataFrame*) – Input dataset.

Returns

- **pattern_ids** (*numpy.ndarray*) – Pattern ID array.
- **pattern_dict** (*dict*) – Dictionary of patterns indexed by pattern IDs. Contains a pattern and count for each pattern ID.

pyrolite.util.missing.**cooccurence_pattern**(*Y: ndarray[tuple[int, int]]*, *normalize: bool = False*, *log: bool = False*) → ndarray[tuple[int, int]]

Get the co-occurence patterns from an array.

Parameters

- **Y** (*numpy.ndarray | pandas.DataFrame*) – Input dataset.
- **normalize** (*bool*) – Whether to normalize the cooccurence to compare disparate variables.
- **log** (*bool*) – Whether to take the log of the cooccurence.

Returns
co_occur – Cooccurence frequency array.

Return type
numpy.ndarray

pyrolite.util.units

pyrolite.util.units.**scale**(*in_unit*, *target_unit='ppm'*)

Provides the scale difference between two mass units.

Parameters

- **in_unit** (*str*) – Units to be converted from
- **target_unit** (*str*) – Units to scale to.

Todo

- Implement different inputs: str, list, pandas.Series

Return type
float

pyrolite.util.types

pyrolite.util.types.**iscollection**(*obj*) → bool

Checks whether an object is an iterable collection.

Parameters
obj – Object to check.

Returns
Boolean indication of whether the object is a collection.

Return type
bool

pyrolite.util.meta

pyrolite.util.meta.**get_module_datafolder**(*module: str = 'pyrolite'*, *subfolder: str | None = None*) → Path

Returns the path of a module data folder.

Parameters
subfolder (*str*) – Subfolder within the module data folder.

Return type
pathlib.Path

pyrolite.util.meta.**pyrolite_datafolder**(*subfolder: str | None = None*) → Path

Returns the path of the pyrolite data folder.

Parameters
subfolder (*str*) – Subfolder within the pyrolite data folder.

Return type
pathlib.Path

pyrolite.util.meta.**take_me_to_the_docs**()

Opens the pyrolite documentation in a webbrowser.

pyrolite.util.meta.**sphinx_doi_link**(*doi: str*) → str

Generate a string with a restructured text link to a given DOI.

Parameters
doi (*str*)

Returns
String with doi link.

Return type
str

pyrolite.util.meta.**subkwargs**(*kwargs: dict*, **f: Callable*) → dict

Get a subset of keyword arguments which are accepted by a function.

Parameters

- **kwargs** (*dict*) – Dictionary of keyword arguments.
- **f** (*Callable*) – Function(s) to check.

Returns
Dictionary containing only relevant keyword arguments.

Return type
dict

pyrolite.util.meta.**inargs**(*name: str*, **funcs: Callable*) → bool

Check if an argument is a possible input for a specific function.

Parameters

- **name** (*str*) – Argument name.
- **f** (*Callable*) – Function(s) to check.

Return type
bool

pyrolite.util.meta.**update_docstring_references**(*obj*, *ref: str = 'ref'*)

Updates docstring reference names to strings including the function name. Decorator will return the same function with a modified docstring. Sphinx likes unique names - specifically for citations, not so much for footnotes.

Parameters

- **obj** (*func | class*) – Class or function for which to update documentation references.
- **ref** (*str*) – String to replace with the object name.

Returns
Object with modified docstring.

Return type
func | class

pyrolite.util.skl

Utilities for use with `sklearn`.

>

pyrolite.util.skl.vis

pyrolite.util.skl.vis.**plot_confusion_matrix**(**args*, *ax: Axes | None = None*, *classes: list[str] | None = None*, *class_order: list[str] | None = None*, *normalize: bool = False*, *title: str = 'Confusion Matrix'*, *cmap: str | Colormap = <matplotlib.colors.LinearSegmentedColormap object>*, *norm: Norm | None = None*, *xlabelrotation: float | None = None*, *fmt: str | None = None*) → Axes

This function prints and plots the confusion matrix. Normalization can be applied by setting *normalize=True*.

Parameters

- **args** (*tuple*) – Data to evaluate and visualise a confusion matrix:
  - A single confusion matrix (n x n)
  - A tuple of (y_test, y_predict)
  - A tuple of (classifier_model, X_test, y_test)
- **ax** (*matplotlib.axes.Axes*) – Axis to plot on, if one exists.
- **classes** (*list*) – List of class names to use as labels, and for ordering (see below). This should match the order contained within the model. Where a classifier model is passed, the classes will be directly extracted.

- **class_order** (*list*) – List of classes in the desired order along the axes. Should match the supplied classes where classes are given, or integer indicies for where no named classes are given.
- **normalize** (*bool*) – Whether to normalize the counts for the confusion matrix to the sum of all cases (i.e. be between 0 and 1).
- **title** (*str*) – Title for the axes.
- **cmap** (*str | matplotlib.color.Colormap*) – Colormap for the visualisation of the confusion matrix.
- **norm** (*bool*) – Normalization for the colormap visualisation across the confusion matrix.
- **xlabelrotation** (*float*) – Rotation in degrees for the xaxis labels.
- **fmt** (*str*) – Format to use for filling values.

Returns
ax

Return type
matplotlib.axes.Axes

pyrolite.util.skl.vis.**plot_gs_results**(*gs: GridSearchCV*, *xvar: str | None = None*, *yvar: str | None = None*) → Axes

Plots the results from a GridSearch showing location of optimum in 2D.

pyrolite.util.skl.vis.**alphas_from_multiclass_prob**(*probs: ndarray[tuple[int, int]]*, *method: str = 'entropy'*, *alpha: float = 1.0*) → ndarray[tuple[int], dtype[floating]]

Take an array of multiclass probabilities and map to an alpha variable.

Parameters

- **probs** (*numpy.ndarray*) – Multiclass probabilities with shape (nsamples, nclasses).
- **method** (str, *entropy | kl_div*) – Method for mapping probabilities to alphas.
- **alpha** (*float*) – Optional specification of overall maximum alpha value.

Returns
a – Alpha values for each sample with shape (nsamples, 1).

Return type
numpy.ndarray

pyrolite.util.skl.vis.**plot_mapping**(*X: ndarray[tuple[int, int]]*, *Y: ndarray[tuple[int, int]] | ndarray[tuple[int]] | BaseEstimator*, *mapping: ndarray[tuple[int, int]] | TransformerMixin | None = None*, *ax: Axes | None = None*, *cmap: str | Colormap | None = None*, *alpha: float = 1.0*, *s: float = 10*, *alpha_method: str = 'entropy'*, ***kwargs*) → Axes

Parameters

- **X** (*numpy.ndarray*) – Coordinates in multidimensional space.
- **Y** (*numpy.ndarray | sklearn.base.BaseEstimator*) – An array of targets, or a method to obtain such an array of targets via *Y.predict*. Transformers with probabilistic output (via *Y.predict_proba*) will have these probability estimates accounted for via the alpha channel.
- **mapping** (numpy.ndarray | *~sklearn.base.TransformerMixin*) – Mapped points or transformer to create mapped points.

- **ax** (*matplotlib.axes.Axes*) – Axes to plot on.
- **cmap** (*matplotlib.cm.ListedColormap*) – Colormap to use for the classification visualisation (ideally this should be a discrete colormap unless the classes are organised).
- **alpha** (*float*) – Coefficient for alpha.
- **alpha_method** (*'entropy' or 'kl_div'*) – Method to map class probabilities to alpha. *'entropy'* uses a measure of entropy relative to null-scenario of equal distribution across classes, while *'kl_div'* calculates the information gain relative to the same null-scenario.

Returns

- **ax** (*matplotlib.axes.Axes*) – Axes on which the mapping is plotted.
- **tfm** (*sklearn.base.BaseEstimator*) – Fitted mapping transform.

Todo

- Option to generate colors for individual classes

  This could be based on the distances between their centres in multidimensional space (or low dimensional mapping of this space), enabling a continuous (n-dimensional) colormap to be used to show similar classes, in addition to classification confidence.

>

pyrolite.util.skl.pipeline

pyrolite.util.skl.pipeline.**fit_save_classifier**(*clf: RegressorMixin | ClassifierMixin*, *X_train: ndarray[tuple[int, int]] | DataFrame*, *y_train: ndarray[tuple[int]] | Series*, *directory: str | Path = '.'*, *name: str = 'clf'*, *extension: str = '.joblib'*) → BaseEstimator

Fit and save a classifier model. Also save relevant metadata where possible.

Parameters

- **clf** (*sklearn.base.BaseEstimator*) – Classifier or gridsearch.
- **X_train** (*numpy.ndarray | pandas.DataFrame*) – Training data.
- **y_train** (*numpy.ndarray | pandas.Series*) – Training true classes.
- **directory** (*str | pathlib.Path*) – Path to the save directory.
- **name** (*str*) – Name of the classifier.
- **extension** (*str*) – Extension to give the saved classifier pickled witih joblib.

Returns

clf – Fitted classifier.

Return type

sklearn.base.BaseEstimator

pyrolite.util.skl.pipeline.**classifier_performance_report**(*clf: RegressorMixin | ClassifierMixin*, *X_test: ndarray[tuple[int, int]] | DataFrame*, *y_test: ndarray[tuple[int]] | Series*, *classes: list[str] | None = None*, *directory: str | Path = '.'*, *name: str = 'clf'*)

Output a performance report for a classifier. Currently outputs the overall classification score, a confusion matrix and where relevant an indication of variation seen across the gridsearch (currently only possible for 2D searches).

Parameters

- **clf** (sklearn.base.BaseEstimator | *sklearn.model_selection.GridSearchCV*) – Classifer or gridsearch.
- **X_test** (*numpy.ndarray* | *pandas.DataFrame*) – Input data for testing.
- **y_test** (*numpy.ndarray* | *pandas.Series*) – Labelled/target data for testing.
- **classes** (*list*) – Names of classes.
- **directory** (*str* | *pathlib.Path*) – Path to the save directory.
- **name** (*str*) – Name of the classifier.

Returns

clf – Fitted classifier.

Return type

sklearn.base.BaseEstimator

pyrolite.util.skl.pipeline.**SVC_pipeline**(*sampler: TransformerMixin | None = None*, *balance: bool = True*, *transform: TransformerMixin | None = None*, *scaler: TransformerMixin | None = None*, *kernel: str | Callable = 'rbf'*, *decision_function_shape: str = 'ovo'*, *probability: bool = False*, *cv: BaseCrossValidator | None = None*, *param_grid: dict | None = None*, *n_jobs: int = 4*, *verbose: int = 10*, *cache_size: float = 500*, ***kwargs*) → GridSearchCV

A convenience function for constructing a Support Vector Classifier pipeline.

Parameters

- **sampler** (*sklearn.base.TransformerMixin*) – Resampling transformer.
- **balance** (*bool*) – Whether to balance the class weights for the classifier.
- **transform** (*sklearn.base.TransformerMixin*) – Preprocessing transformer.
- **scaler** (*sklearn.base.TransformerMixin*) – Scale transformer.
- **kernel** (*str* | *Callable*) – Name of kernel to use for the support vector classifier (*'linear'|'rbf'|'poly'|'sigmoid'*). Optionally, a custom kernel function can be supplied (see `sklearn` docs for more info).
- **decision_function_shape** (*str*) – Shape of the decision function surface. *'ovo'* one-vs-one classifier of libsvm (returning classification of shape *(samples, classes*(classes-1)/2)*), or the default *'ovr' one-vs-rest classifier which will return classification estimation shape of `(samples, classes)`.*
- **probability** (*bool*) – Whether to use a calibrated classifier to estimate probablities.
- **cv** (*int* | *sklearn.model_selection.BaseSearchCV*) – Cross validation search. If an integer *k* is provided, results in default *k*-fold cross validation. Optionally, if a

sklearn.model_selection.BaseSearchCV instance is provided, it will be used directly (enabling finer control, e.g. over sorting/shuffling etc).

- **param_grid** (*dict*) – Dictionary reprenting a parameter grid for the support vector classifier. Typically contains 1D arrays of grid indicies for SVC parameters each prefixed with *svc__* (e.g. *dict(svc__gamma=np.logspace(-1, 3, 5), svc__C=np.logspace(-0.5, 2, 5))*.
- **n_jobs** (*int*) – Number of processors to use for the SVC construction. Note that providing *n_jobs = -1* will use all available processors.
- **verbose** (*int*) – Level of verbosity for the pipeline logging output.
- **cache_size** (*float*) – Specify the size of the kernel cache (in MB).

Returns

gs – Gridsearch object containing the results of the SVC training across the parameter grid. Access the best estimator with *gs.best_estimator_* and its parameters with *gs.best_params_*.

Return type

sklearn.model_selection.GridSearchCV

Notes

See also: *sklearn.svm.SVC*

class pyrolite.util.skl.pipeline.**PdUnion**(*estimators: list | None = None*)

fit(*X: ndarray[tuple[int, int]] | DataFrame*, *y: ndarray[tuple[int]] | Series | None = None*)

transform(*X: ndarray[tuple[int, int]] | DataFrame*)

>

pyrolite.util.skl.select

class pyrolite.util.skl.select.**TypeSelector**(*dtype: str | dtype*)

fit(*X: DataFrame*, *y: Series | None = None*)

transform(*X*)

class pyrolite.util.skl.select.**ColumnSelector**(*columns: list[str]*)

fit(*X: DataFrame*, *y: Series | None = None*)

transform(*X: DataFrame*)

class pyrolite.util.skl.select.**CompositionalSelector**(*columns: list[str] | None = None*, *inverse: bool = False*)

fit(*X: DataFrame*, *y: Series | None = None*)

transform(*X: DataFrame*)

class pyrolite.util.skl.select.**MajorsSelector**(*components: list[str] | None = None*)

fit(*X: DataFrame*, *y: Series | None = None*)

transform(*X: DataFrame*)

class pyrolite.util.skl.select.**ElementSelector**(*components: list[str] | None = None*)

fit(*X: DataFrame*, *y: Series | None = None*)

transform(*X*)

class pyrolite.util.skl.select.**REESelector**(*components: list[str] | None = None*)

fit(*X: DataFrame*, *y: Series | None = None*)

transform(*X: DataFrame*)

>

pyrolite.util.skl.transform

class pyrolite.util.skl.transform.**DropBelowZero**(***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X*, **args*)

class pyrolite.util.skl.transform.**LinearTransform**(***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*)

class pyrolite.util.skl.transform.**ExpTransform**(***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X*, **args*)

class pyrolite.util.skl.transform.**LogTransform**(*fmt_string: str = 'Ln({})'*, ***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*)

class pyrolite.util.skl.transform.**ALRTransform**(*label_mode: str = 'numeric'*, ***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

class pyrolite.util.skl.transform.**CLRTransform**(*label_mode: str = 'numeric'*, ***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X*, **args*, ***kwargs*)

class pyrolite.util.skl.transform.**ILRTransform**(*label_mode: str = 'numeric'*, ***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

class pyrolite.util.skl.transform.**SphericalCoordTransform**(***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

class pyrolite.util.skl.transform.**BoxCoxTransform**(***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

inverse_transform(*Y: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*, **args*, ***kwargs*)

fit(*X*, **args*, ***kwargs*)

class pyrolite.util.skl.transform.**Devolatilizer**(*exclude: list[str] | None = None*, *renorm: bool = True*)

fit(*X: DataFrame | Series*, **args*, ***kwargs*)

transform(*X: DataFrame | Series | ndarray[tuple[int, int]] | ndarray[tuple[int]]*)

class pyrolite.util.skl.transform.**ElementAggregator**(*renorm: bool = True*, *form: str = 'oxide'*)

fit(*X: DataFrame | Series*, **args*, ***kwargs*)

transform(*X: DataFrame | Series*)

class pyrolite.util.skl.transform.**LambdaTransformer**(*norm_to: str = 'Chondrite_PON'*, *exclude: list[str] | None = None*, *params: list | None = None*, *degree: int = 5*)

fit(*X: DataFrame | Series*, **args*, ***kwargs*)

transform(*X: DataFrame | Series*)

>

pyrolite.util.skl.impute

pyrolite.util.classification

Utilities for rock chemistry and mineral abundance classification.

Todo

- Petrological classifiers: QAPF (aphanitic/phaneritic), gabbroic Pyroxene-Olivine-Plagioclase, ultramafic Olivine-Orthopyroxene-Clinopyroxene

class pyrolite.util.classification.**PolygonClassifier**(*name: str | None = None*, *axes: dict | None = None*, *fields: dict | None = None*, *scale: float = 1.0*, *transform: Callable | str | None = None*, *mode: str | None = None*, ***kwargs*)

A classifier model built form a series of polygons defining specific classes.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*dict*) – Mapping from plot axes to variables to be used for labels.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.
- **scale** (*float*) – Default maximum scale for the axes. Typically 100 (wt%) or 1 (fractional).
- **xlim** (*tuple*) – Default x-limits for this classifier for plotting.
- **ylim** (*tuple*) – Default y-limits for this classifier for plotting.

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

class pyrolite.util.classification.**TAS**(*which_model: str | None = None*, ***kwargs*)

Total-alkali Silica Diagram classifier from Middlemost (1994) **[#pyrolite.util.classification.TAS_1]_**, a closed-polygon variant after Le Bas et al. (1992) **[#pyrolite.util.classification.TAS_2]_**.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.
- **scale** (*float*) – Default maximum scale for the axes. Typically 100 (wt%) or 1 (fractional).
- **xlim** (*tuple*) – Default x-limits for this classifier for plotting.
- **ylim** (*tuple*) – Default y-limits for this classifier for plotting.
- **which_model** (*str*) – The name of the model variant to use, if not Middlemost.

References

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 100.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, *label_at_centroid: bool = True*, ***kwargs*) → Axes

Add the TAS fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which labels to add to the polygons (e.g. for TAS, 'volcanic', 'intrusive' or the field 'ID').
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.
- **label_at_centroid** (*bool*) – Whether to label the fields at the centroid (True) or at the visual center of the field (False).

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: **list[str]**

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**USDASoilTexture**(***kwargs*)

United States Department of Agriculture Soil Texture classification model [#pyrolite.util.classification.USDASoilTexture_1]_ [#pyrolite.util.classification.USDASoilTexture_2]_.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.

References

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property **axis_components**: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**QAP**(***kwargs*)

IUGS QAP ternary classification [#pyrolite.util.classification.QAP_1]_ [#pyrolite.util.classification.QAP_2]_.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.

References

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: **list[str]**

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**FeldsparTernary**(***kwargs*)

Simplified feldspar diagram classifier, based on a version printed in the second edition of 'An Introduction to the Rock Forming Minerals' (Deer, Howie and Zussman).

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.
- **mode** (*str*) – Mode of the diagram to use; two are currently available - 'default', which fills the entire ternary space, and 'miscibility-gap' which gives a simplified approximation of the miscibility gap.

References

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.PeralkalinityClassifier

predict(*df: DataFrame*) → Series

class pyrolite.util.classification.JensenPlot(***kwargs*)

Jensen Plot for classification of subalkaline volcanic rocks [#pyrolite.util.classification.JensenPlot_1]_.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.

References

Notes

Diagram used for the classification classification of subalkalic volcanic rocks. The diagram is constructed for molar cation percentages of Al, Fe+Ti and Mg, on account of these elements' stability upon metamorphism. This particular version uses updated labels relative to Jensen (1976), in which the fields have been extended to the full range of the ternary plot.

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.

- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, [‘F’, ‘T1’] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**SpinelTrivalentTernary**(***kwargs*)

Spinel Trivalent Ternary classification - designed for data in atoms per formula unit

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing ‘name’ and ‘fields’ items.

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field ‘name’ or ‘ID’.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, [‘F’, ‘T1’] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**SpinelFeBivariate**(***kwargs*)

Fe-Spinel classification, designed for data in atoms per formula unit.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**Pettijohn**(***kwargs*)

Pettijohn (1973) sandstones classification **[#pyrolite.util.classification.Pettijohn_1]_**.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.

References

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**Herron**(***kwargs*)

Herron (1988) sandstones classification **[#pyrolite.util.classification.Herron_1]_**.

Parameters

- **name** (*str*) – A name for the classifier model.
- **axes** (*list | tuple*) – Names of the axes corresponding to the polygon coordinates.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.

References

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels for the polygons.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object. (e.g. for TAS, ['F', 'T1'] to plot the Foidite and Trachyte fields). An empty list corresponds to plotting all the polygons.

Returns
ax

Return type
matplotlib.axes.Axes

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Predict the classification of samples using the polygon-based classifier.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Series containing classifer predictions. If a dataframe was input, it inherit the index.

Return type
pandas.Series

class pyrolite.util.classification.**Magnetite**(*which_model: str | None = None*, ***kwargs*)

Magnetite trace element discrimination diagrams for mineral deposit types from Dupuis and Beaudoin (2011) [#pyrolite.util.classification.Magnetite_1]_. Fields are defined on log-log axes for concentrations in wt%.

Parameters

- **which_model** (*str*) – Diagram to use: 'TiV_CaAlMn' (Ca+Al+Mn vs Ti+V; default) or 'TiV_NiCrMn' (Ni/(Cr+Mn) vs Ti+V).
- **name** (*str*) – A name for the classifier model.
- **axes** (*dict*) – Mapping from plot axes to variables to be used for labels.
- **fields** (*dict*) – Dictionary describing indiviudal polygons, with identifiers as keys and dictionaries containing 'name' and 'fields' items.
- **xlim** (*tuple*) – Default x-limits for this classifier for plotting.
- **ylim** (*tuple*) – Default y-limits for this classifier for plotting.

References

property axis_components: list[str]

Get the axis components used by the classifier.

Returns
Ordered names for axes used by the classifier.

Return type
tuple

models = ('TiV_CaAlMn', 'TiV_NiCrMn')

predict(*X: ndarray[tuple[int, int], dtype[number]] | DataFrame*, *data_scale=None*) → Series

Classify samples; point-in-polygon tests are done in log10 space, where the field boundaries are straight lines.

Parameters

- **X** (*numpy.ndarray | pandas.DataFrame*) – Data to classify (wt%). Dataframes must contain the axis components.
- **data_scale** (*float*) – Maximum scale for the data. Typically 100 (wt%) or 1 (fractional).

Returns
Classifier predictions; 'none' for samples outside all fields.

Return type
pandas.Series

add_to_axes(*ax: Axes | None = None*, *fill: bool = False*, *axes_scale: float = 1.0*, *add_labels: bool = False*, *which_labels: str = 'ID'*, *which_ids: list[str] | None = None*, ***kwargs*) → Axes

Add the fields from the classifier to an axis, with log-scaled axes.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Axis to add the polygons to.
- **fill** (*bool*) – Whether to fill the polygons.
- **axes_scale** (*float*) – Maximum scale for the axes. Typically 100 (for wt%) or 1 (fractional).
- **add_labels** (*bool*) – Whether to add labels at the (log-space) polygon centroids.
- **which_labels** (*str*) – Which data to use for field labels - field 'name' or 'ID'.
- **which_ids** (*list*) – List of field IDs corresponding to the polygons to add to the axes object.

Returns
ax

Return type
matplotlib.axes.Axes

pyrolite.util.log

pyrolite.util.log.**Handle**(*logger: Logger | str*, *handler_class: str = <class 'logging.StreamHandler'>*, *formatter: str = '%(asctime)s %(name)s - %(levelname)s: %(message)s'*, *level: str | int | None = None*) → Logger

Handle a logger with a standardised formatting.

Parameters

- **logger** (*logging.Logger | str*) – Logger or module name to source a logger from.
- **handler_class** (*logging.Handler*) – Handler class for the logging messages.
- **formatter** (str | *logging.Formatter*) – Formatter for the logging handler. Strings will be passed to the *logging.Formatter* constructor.
- **level** (*str*) – Logging level for the handler.

Returns
Configured logger.

Return type
logging.Logger

class pyrolite.util.log.**ToLogger**(*logger*, *level=None*)

Output stream which will output to logger module instead of stdout.

buf = ''

logger = None

level = None

write(*buf*)

Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

flush()

Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

See also

Extensions

6.2 Data Examples

6.2.1 Mineral Database

pyrolite includes a limited mineral database which is useful for for looking up endmember compositions. This part of the package is being actively developed, so expect expansions and improvements soon.

See also

Examples:
Mineral Database

Total running time of the script: (0 minutes 0.000 seconds)

6.2.2 Geological Timescale

pyrolite includes a simple geological timescale, based on a recent verion of the International Chronostratigraphic Chart[1]. The *Timescale* class can be used to look up names for specific geological ages, to look up times for known geological age names and to access a reference table for all of these.

```
from pyrolite.util.time import Timescale

ts = Timescale()

eg = ts.data.iloc[:, :5]  # the first five columns of this data table
eg
```

References

[1] Cohen, K.M., Finney, S.C., Gibbard, P.L., Fan, J.-X., 2013. The ICS International Chronostratigraphic Chart. Episodes 36, 199–204.

See also

Examples:
Timescale

Modules, Classes and Functions:
pyrolite.util.time, *Timescale*

Total running time of the script: (0 minutes 0.031 seconds)

6.2.3 Ionic Radii

`pyrolite` incldues a few sets of reference tables for ionic radii in aangstroms (Å) from [Shannon1976] and [WhittakerMuntus1970], each with tables indexed by element, ionic charge and coordination. The easiset way to access these is via the *get_ionic_radii()* function. The function can be used to get radii for individual elements, using a `source` keyword argument to swap between the datasets:

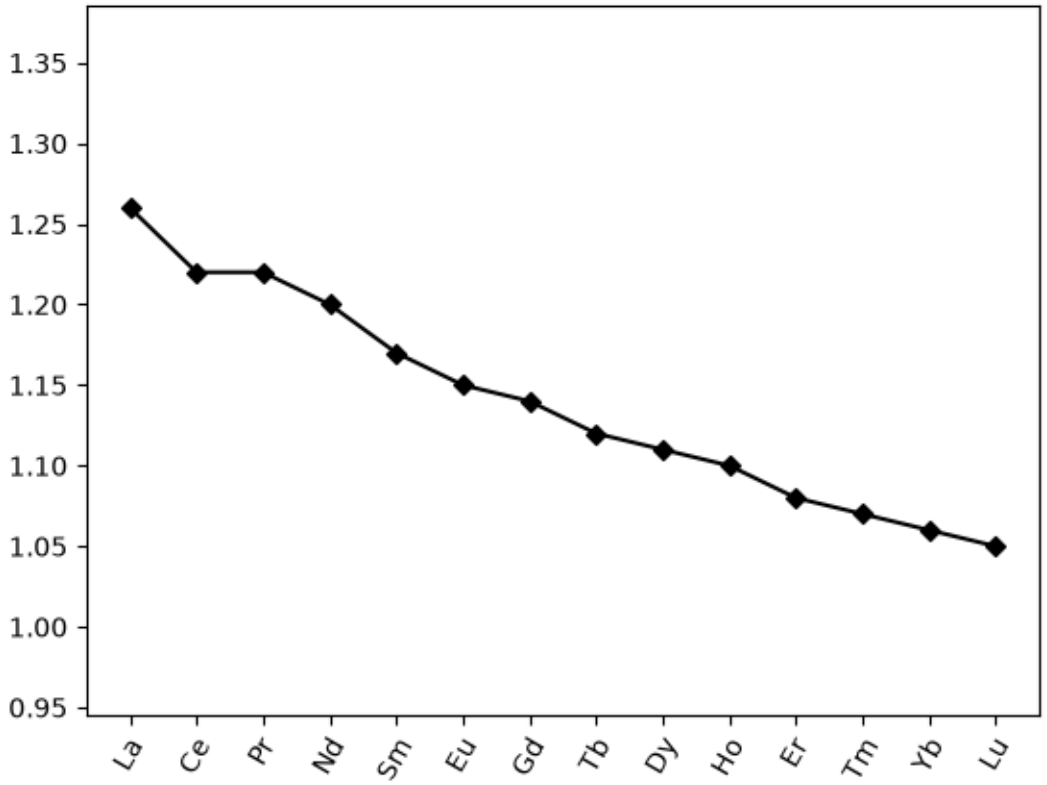
```
import matplotlib.pyplot as plt
import pandas as pd

from pyrolite.geochem.ind import REE, get_ionic_radii

REE_radii = pd.Series(
    get_ionic_radii(REE(), coordination=8, charge=3, source="Whittaker"), index=REE()
)
REE_radii
```

```
La    1.26
Ce    1.22
Pr    1.22
Nd    1.20
Sm    1.17
Eu    1.15
Gd    1.14
Tb    1.12
Dy    1.11
Ho    1.10
Er    1.08
Tm    1.07
Yb    1.06
Lu    1.05
dtype: float64
```

```
REE_radii.pyroplot.spider(color="k", logy=False)
plt.show()
```



References

See also

Examples:
Ionic Radii, lambdas: Parameterising REE Profiles, REE Radii Plot

Functions:
get_ionic_radii(), *pyrolite.geochem.ind.REE()*, *lambda_lnREE()*,

Total running time of the script: (0 minutes 0.062 seconds)

6.2.4 Aitchison Examples

`pyrolite` includes four synthetic datasets which are used in [Aitchison1984] which can be accessed using each of the respective functions `load_boxite()`, `load_coxite()`, `load_hongite()` and `load_kongite()` (all returning a `DataFrame`).

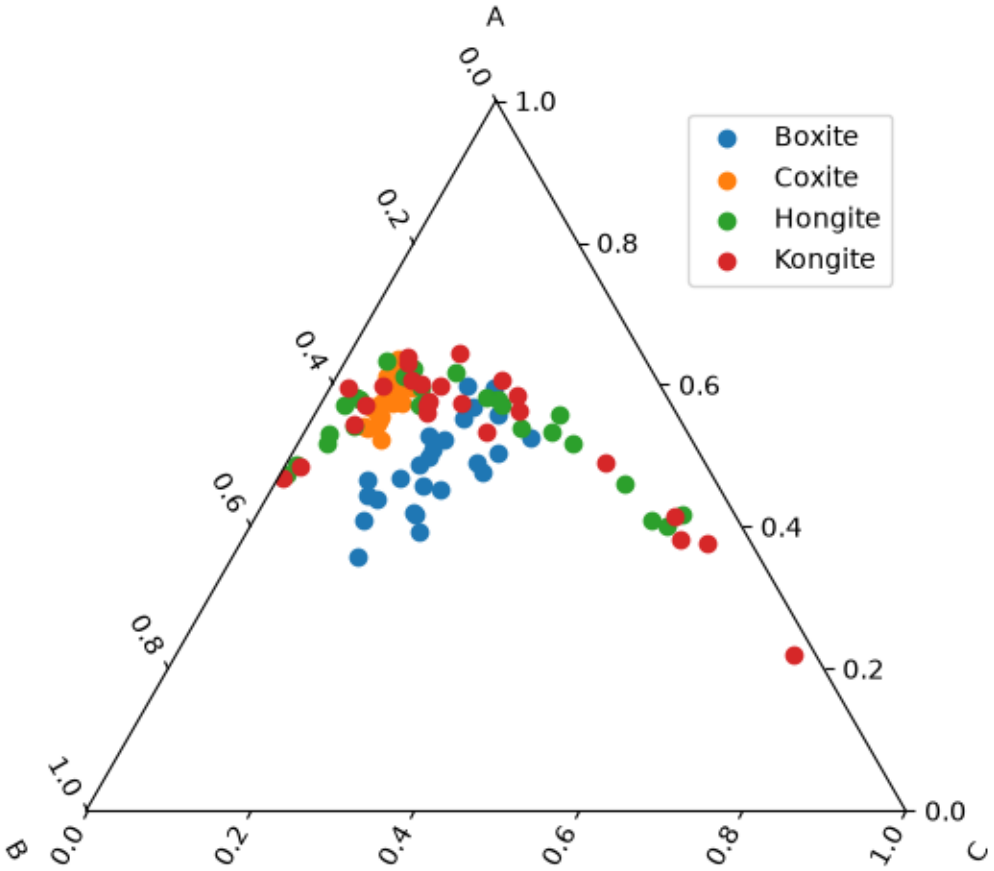
```
from pyrolite.data.Aitchison import load_boxite, load_coxite, load_hongite, load_kongite

df = load_boxite()
df.head()
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1)
for loader in [load_boxite, load_coxite, load_hongite, load_kongite]:
    df = loader()
    ax = df[["A", "B", "C"]].pyroplot.scatter(ax=ax, label=df.attrs["name"])

ax.legend()
plt.show()
```



References

See also

Examples:
Log Ratio Means, Log Transforms, Compositional Data, Ternary Plots

Tutorials:
Ternary Density Plots, Making the Logo

Modules and Functions:
pyrolite.comp.codata, *renormalise()*

Total running time of the script: (0 minutes 0.219 seconds)

6.2.5 Reference Compositions

This page presents the range of compositions within the reference compositon database accessible within `pyrolite`. It's currently a work in progress, but will soon contain extended descriptions and notes for some of the compositions and associated references.

```
import matplotlib.pyplot as plt

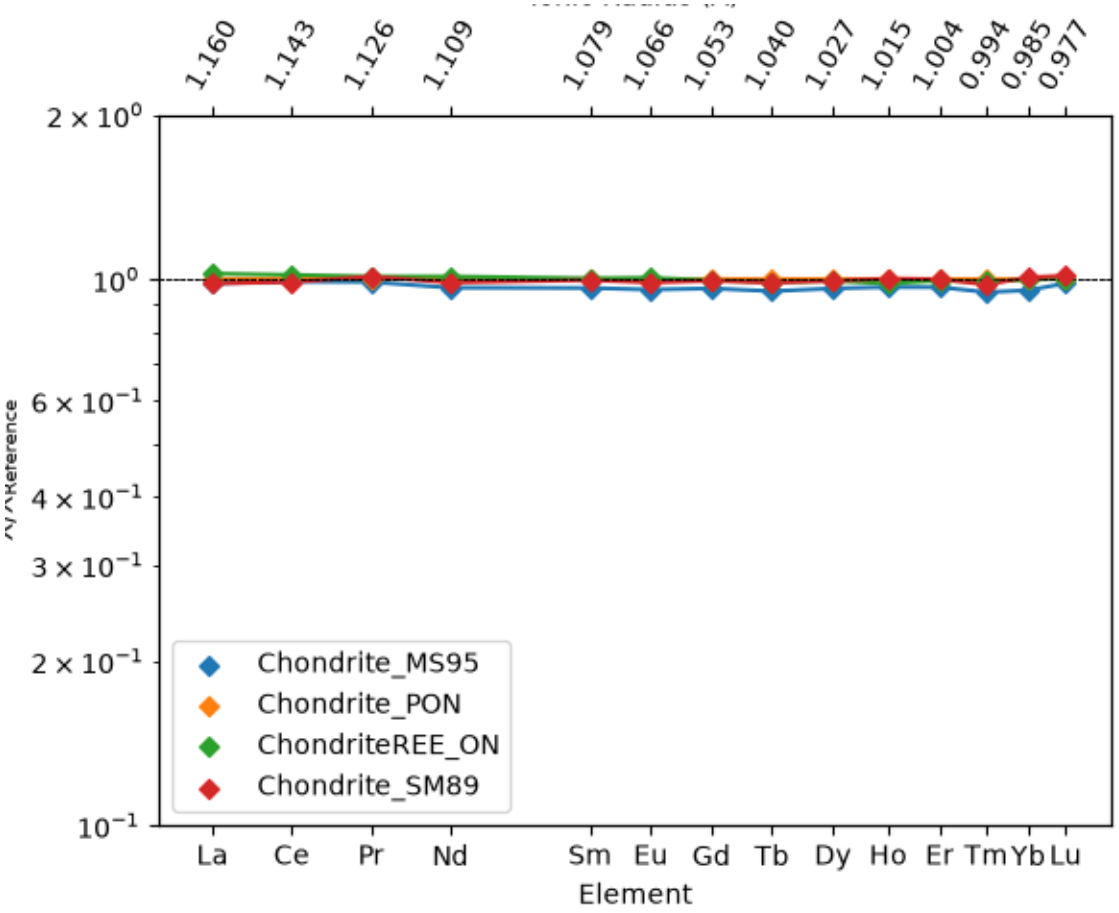
from pyrolite.geochem.norm import all_reference_compositions


refcomps = all_reference_compositions()
norm = "Chondrite_PON"  # a constant composition to normalise to
```

Chondrites

```
fltr = lambda c: c.reservoir == "Chondrite"
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```

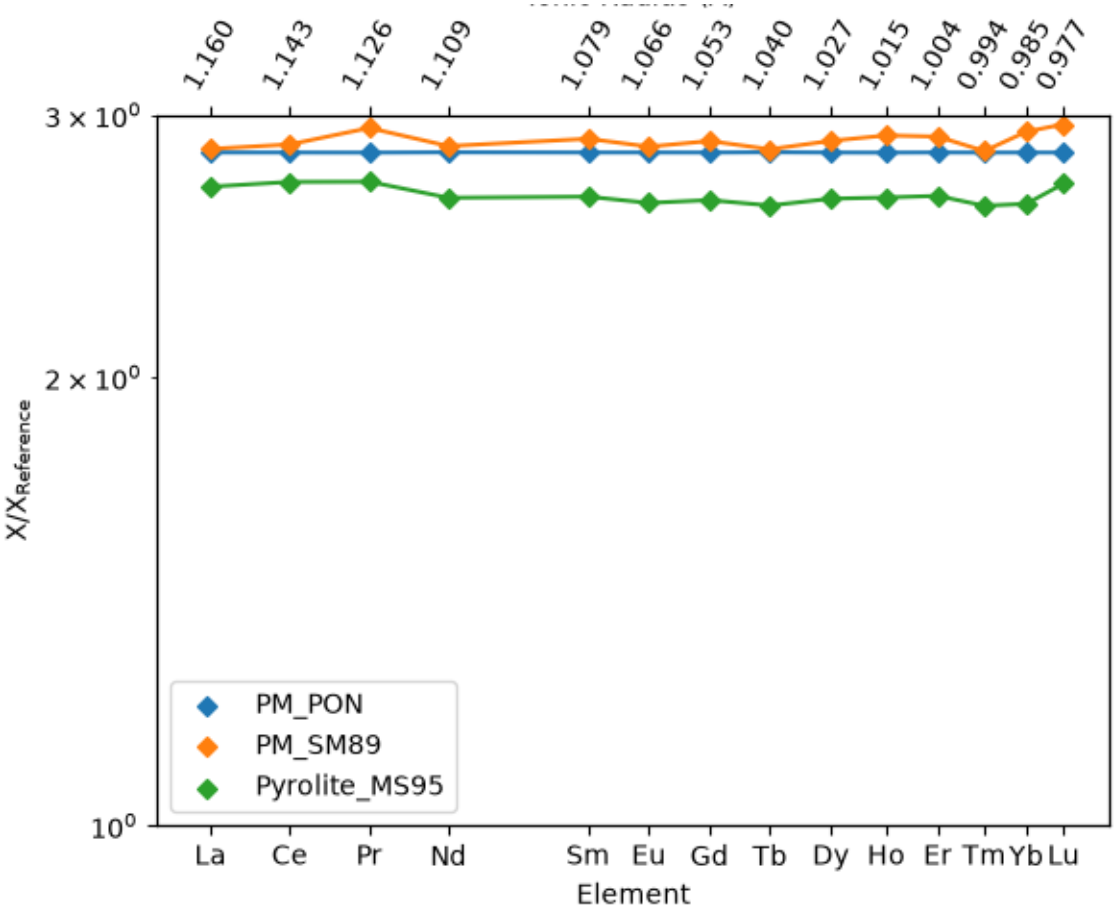


Mantle

Primitive Mantle & Pyrolite

```
fltr = lambda c: c.reservoir in ["PrimitiveMantle", "BSE"]
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

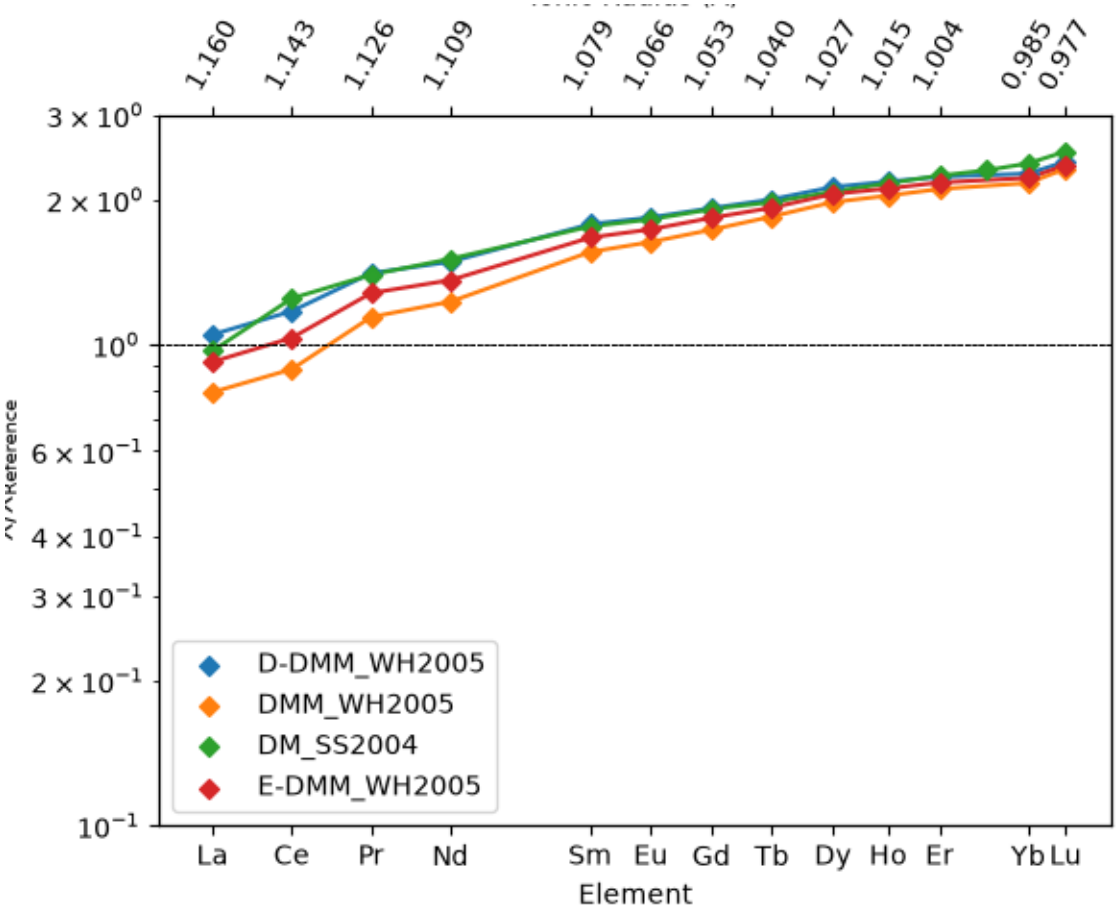
fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Depleted Mantle

```
fltr = lambda c: ("Depleted" in c.reservoir) & ("Mantle" in c.reservoir)
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```

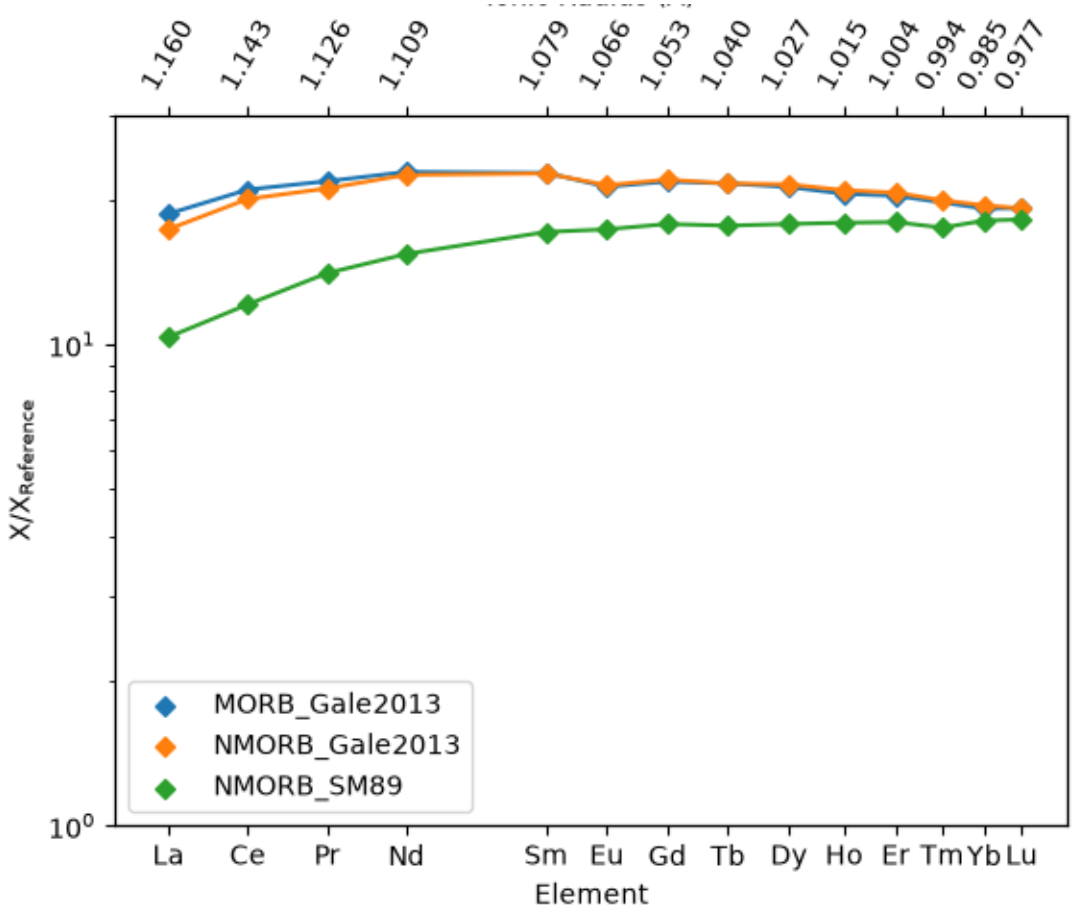


Mid-Ocean Ridge Basalts (MORB)

Average MORB, NMORB

```
fltr = lambda c: c.reservoir in ["MORB", "NMORB"]
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

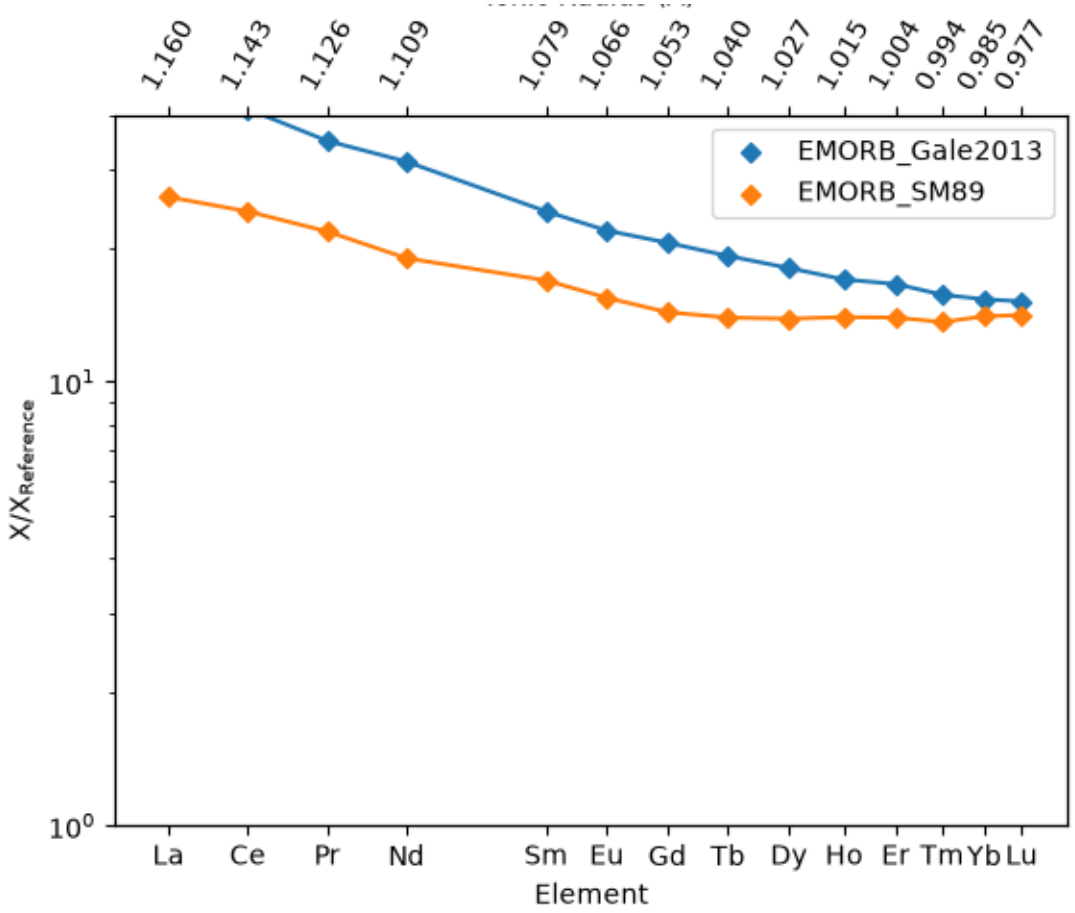
fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Enriched MORB

```
fltr = lambda c: "EMORB" in c.reservoir
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

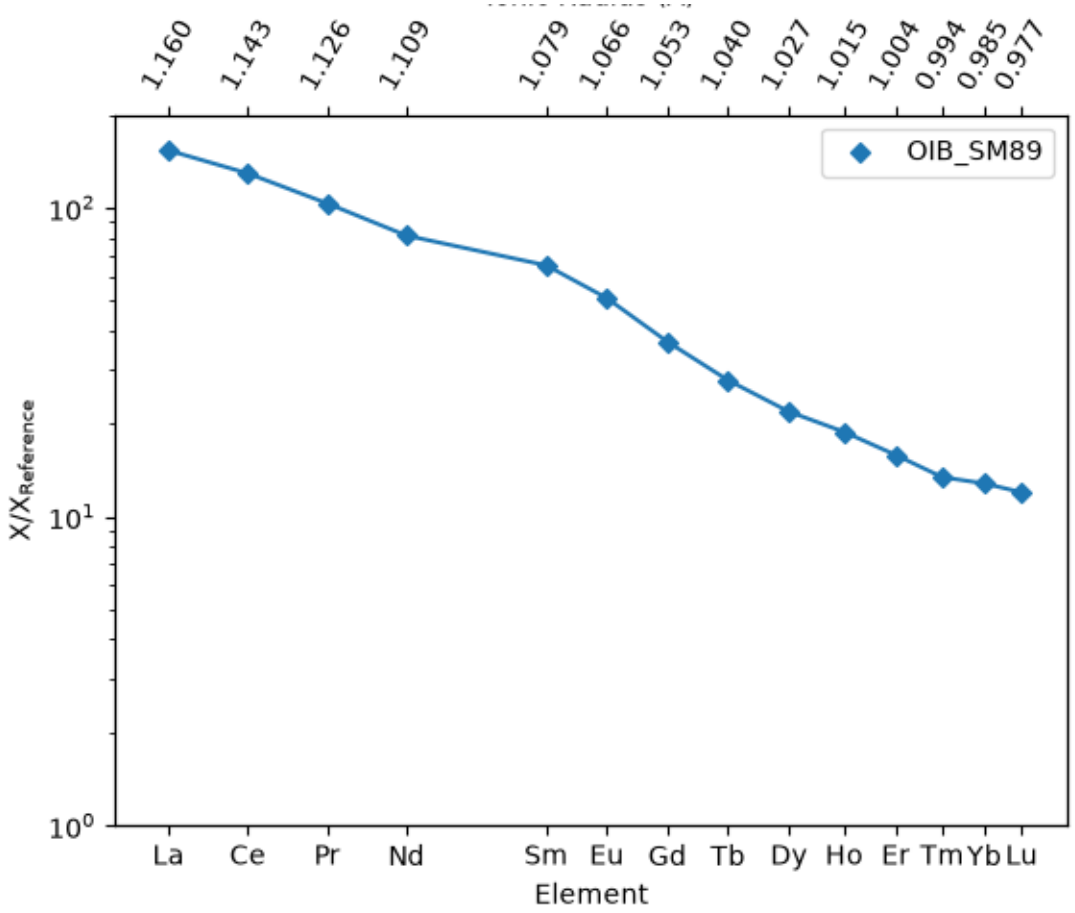
fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Ocean Island Basalts

```
fltr = lambda c: "OIB" in c.reservoir
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```

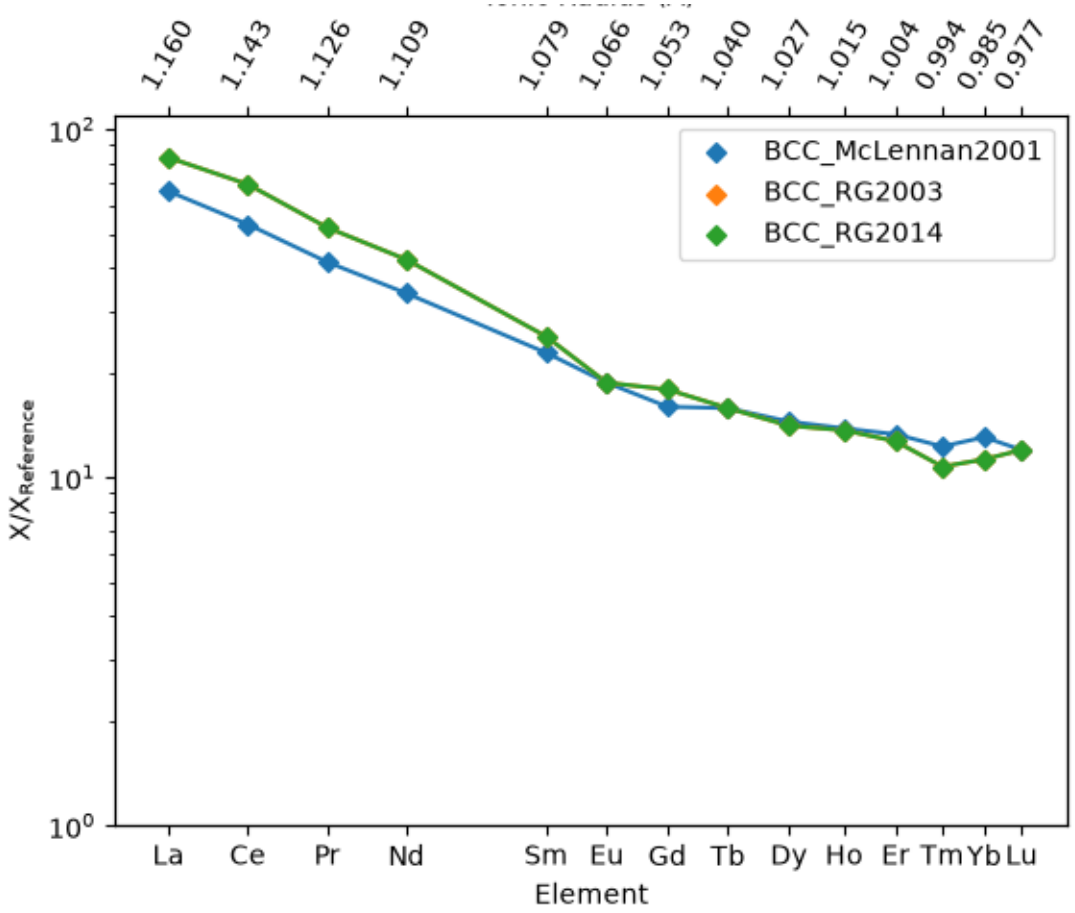


Continental Crust

Bulk Continental Crust

```
fltr = lambda c: c.reservoir == "BulkContinentalCrust"
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

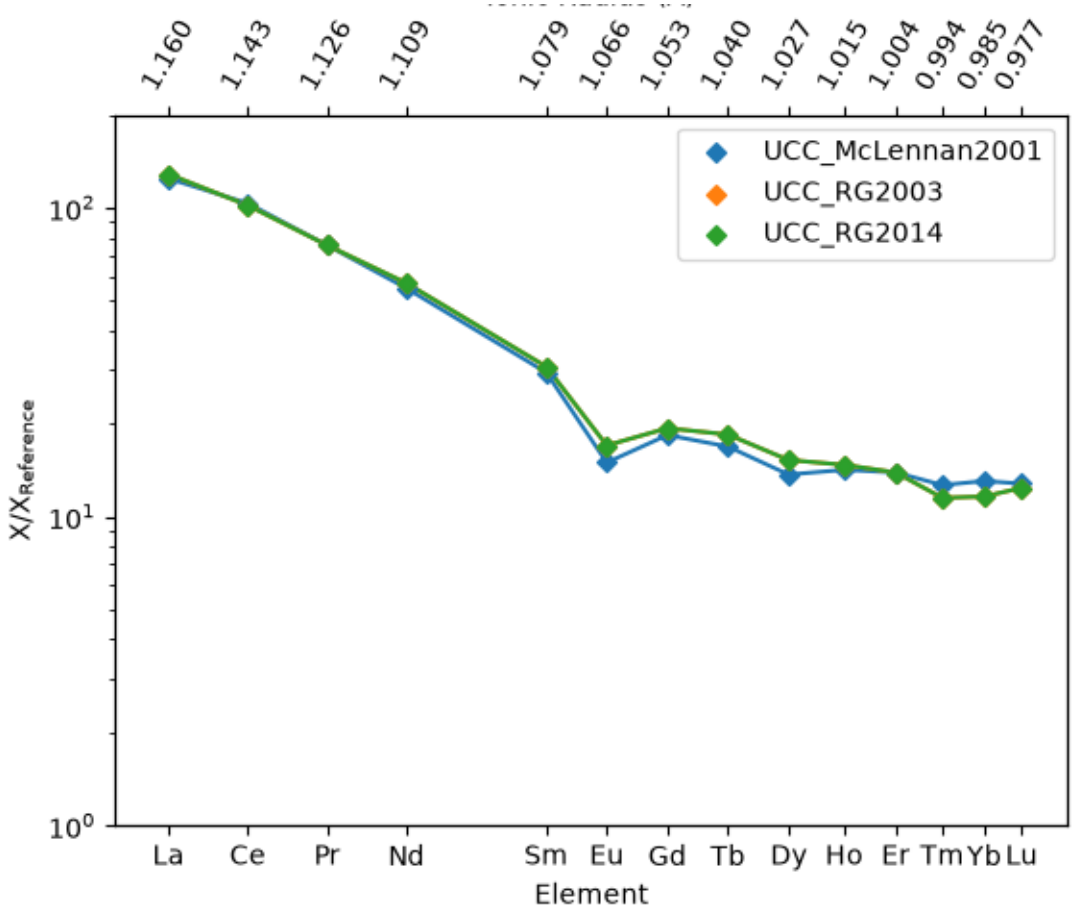
fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Upper Continental Crust

```
fltr = lambda c: c.reservoir == "UpperContinentalCrust"
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

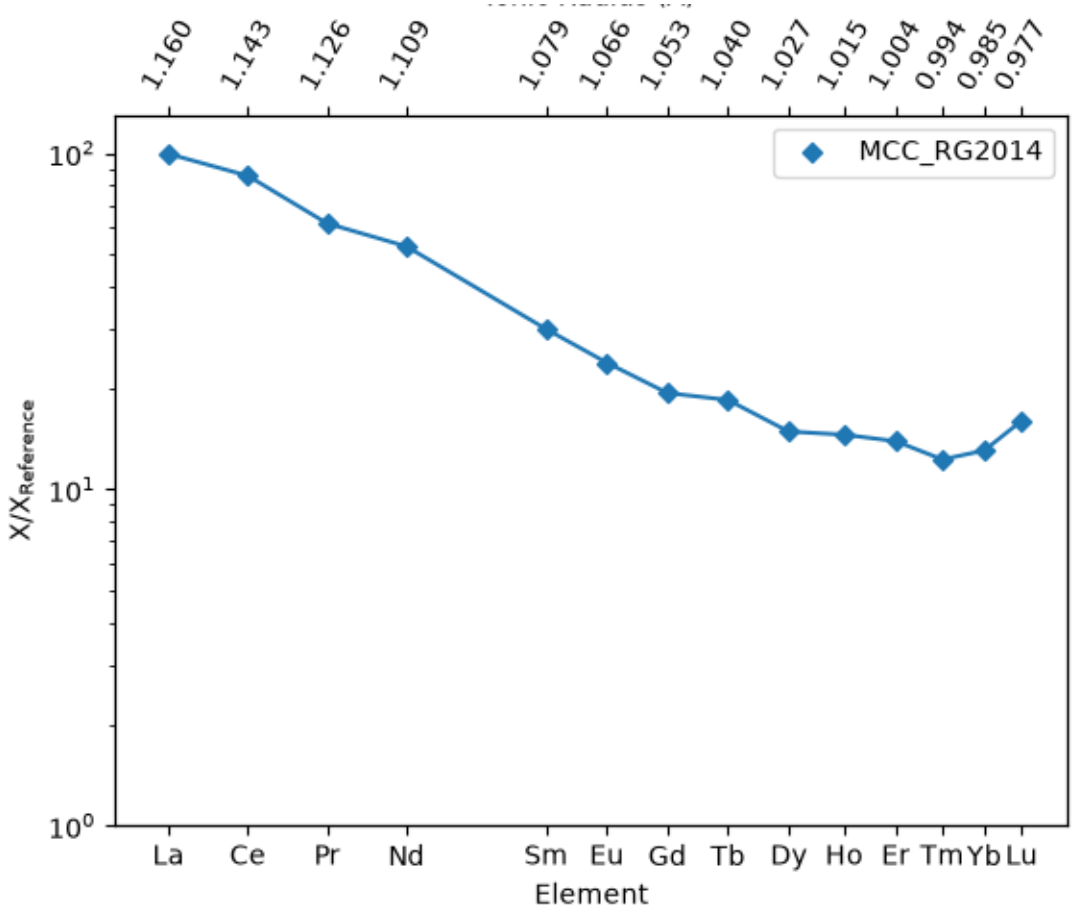
fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Mid-Continental Crust

```
fltr = lambda c: c.reservoir == "MidContinentalCrust"
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

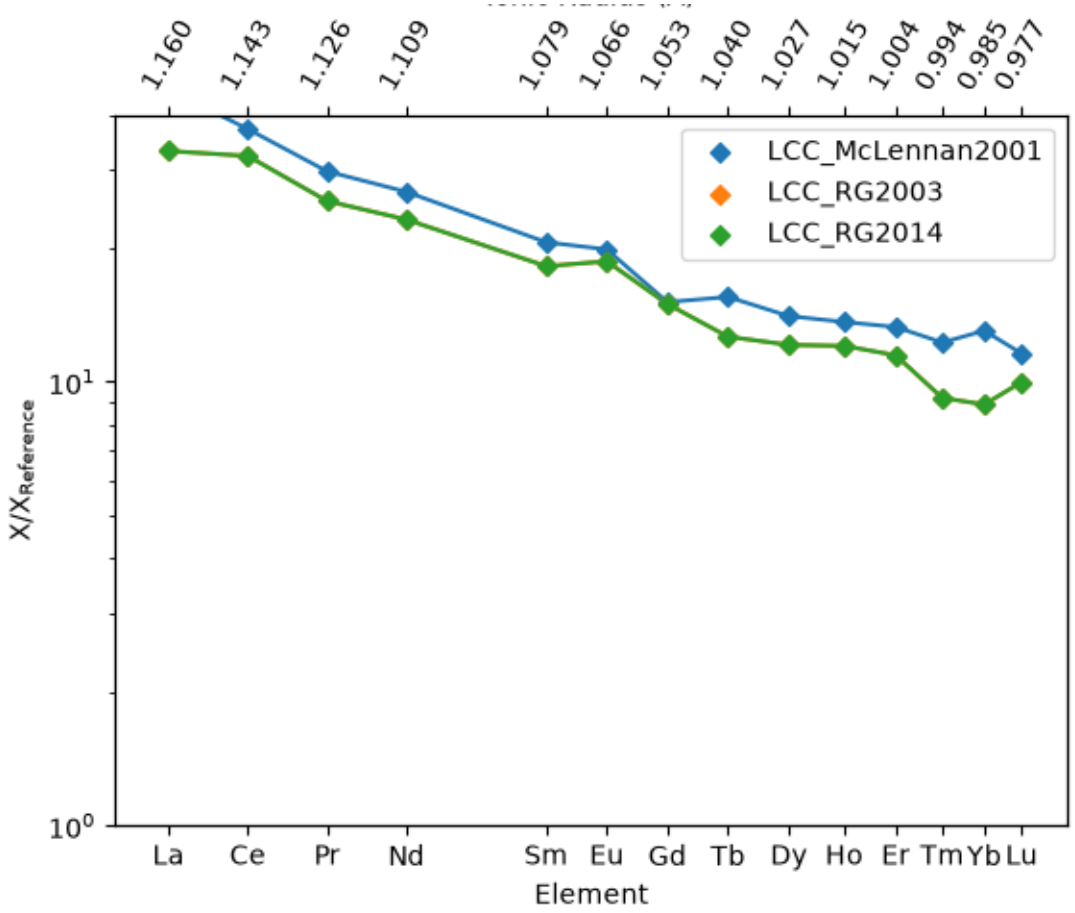
fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Lower Continental Crust

```
fltr = lambda c: c.reservoir == "LowerContinentalCrust"
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

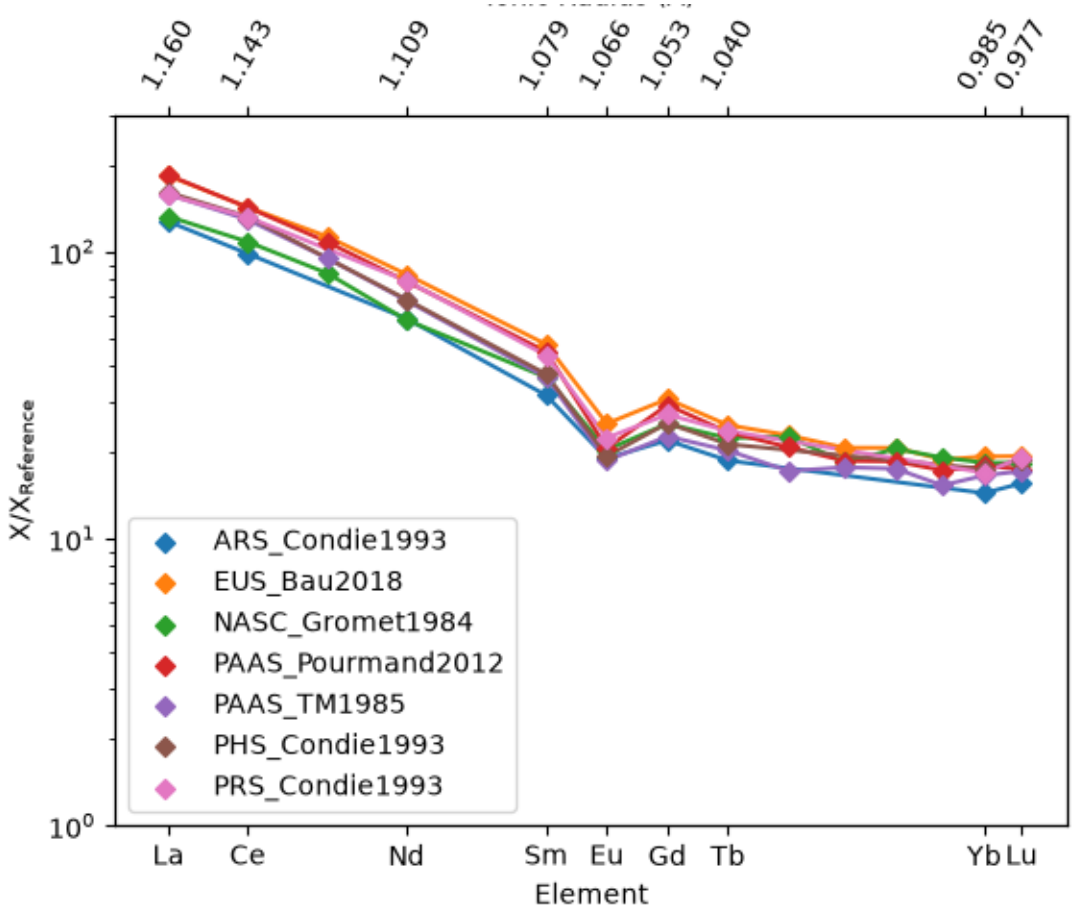
fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Shales

```
fltr = lambda c: "Shale" in c.reservoir
compositions = [x for (name, x) in refcomps.items() if fltr(x)]

fig, ax = plt.subplots(1)
for composition in compositions:
    composition.set_units("ppm")
    df = composition.comp.pyrochem.normalize_to(norm, units="ppm")
    df.pyroplot.REE(unity_line=True, ax=ax, label=composition.name)
ax.legend()
plt.show()
```



Composition List

See also

Examples:
Normalisation

Total running time of the script: (0 minutes 1.917 seconds)

6.3 Extensions

This page will list packages which have been developed as extensions to pyrolite.

Note

This page is a work in progress and will updated soon as submodules from `pyrolite.ext` are migrated out of pyrolite to reduce installation overheads and unnecessary dependencies.

6.3.1 pyrolite-meltsutil

- `pyrolite` extension which wraps the `alphaMELTS` executable
- Functions for working with `alphaMELTS` tables
- Eventually, this will likely instead link to *under-development* python-MELTS.

Note

There are some great things happening on the MELTS-for-scripting front; these utilities will be intended to link your data to these tools.

BIBLIOGRAPHY

[Shannon1976] Shannon RD (1976). Revised effective ionic radii and systematic studies of interatomic distances in halides and chalcogenides. Acta Crystallographica Section A 32:751–767. doi: 10.1107/S0567739476001551.

[WhittakerMuntus1970] Whittaker, E.J.W., Muntus, R., 1970. Ionic radii for use in geochemistry. Geochimica et Cosmochimica Acta 34, 945–956. doi: 10.1016/0016-7037(70)90077-3.

[Pauling1960] Pauling, L., 1960. The Nature of the Chemical Bond. Cornell University Press, Ithaca, NY.

[Cross1902] Cross, W., Iddings, J. P., Pirsson, L. V., & Washington, H. S. (1902). A Quantitative Chemico-Mineralogical Classification and Nomenclature of Igneous Rocks. The Journal of Geology, 10(6), 555–690. doi: 10.1086/621030

[Verma2003] Verma, S. P., Torres-Alvarado, I. S., & Velasco-Tapia, F. (2003). A revised CIPW norm. Swiss Bulletin of Mineralogy and Petrology, 83(2), 197–216.

[Verma2013] Verma, S. P., & Rivera-Gomez, M. A. (2013). Computer Programs for the Classification and Nomenclature of Igneous Rocks. Episodes, 36(2), 115–124.

[LeMaitre1976] Le Maitre, R. W (1976). Some Problems of the Projection of Chemical Data into Mineralogical Classifications. Contributions to Mineralogy and Petrology 56, no. 2 (1 January 1976): 181–89. doi: doi.org/10.1007/BF00399603

[Middlemost1989] Middlemost, Eric A. K. (1989). Iron Oxidation Ratios, Norms and the Classification of Volcanic Rocks. Chemical Geology 77, 1: 19–26. doi: doi.org/10.1016/0009-2541(89)90011-9.

[Shannon1976] Shannon RD (1976). Revised effective ionic radii and systematic studies of interatomic distances in halides and chalcogenides. Acta Crystallographica Section A 32:751–767. doi: 10.1107/S0567739476001551.

[WhittakerMuntus1970] Whittaker, E.J.W., Muntus, R., 1970. Ionic radii for use in geochemistry. Geochimica et Cosmochimica Acta 34, 945–956. doi: 10.1016/0016-7037(70)90077-3.

[Pauling1960] Pauling, L., 1960. The Nature of the Chemical Bond. Cornell University Press, Ithaca, NY.

[Aitchison1984] Aitchison, J., 1984. The statistical analysis of geochemical compositions. Journal of the International Association for Mathematical Geology 16, 531–564. doi: 10.1007/BF01029316

PYTHON MODULE INDEX

p

pyrolite.comp, 227
pyrolite.comp.aggregate, 235
pyrolite.comp.codata, 230
pyrolite.comp.impute, 237
pyrolite.geochem, 204
pyrolite.geochem.alteration, 224
pyrolite.geochem.ind, 212
pyrolite.geochem.ions, 226
pyrolite.geochem.isotope, 227
pyrolite.geochem.magma, 223
pyrolite.geochem.norm, 221
pyrolite.geochem.parse, 222
pyrolite.geochem.transform, 216
pyrolite.mineral, 238
pyrolite.mineral.lattice, 244
pyrolite.mineral.mindb, 246
pyrolite.mineral.normative, 240
pyrolite.mineral.sites, 243
pyrolite.mineral.template, 238
pyrolite.mineral.transform, 243
pyrolite.plot, 189
pyrolite.plot.biplot, 198
pyrolite.plot.color, 203
pyrolite.plot.density, 195
pyrolite.plot.density.grid, 196
pyrolite.plot.density.ternary, 196
pyrolite.plot.parallel, 197
pyrolite.plot.spider, 193
pyrolite.plot.stem, 198
pyrolite.plot.templates, 199
pyrolite.util, 247
pyrolite.util.classification, 297
pyrolite.util.distributions, 281
pyrolite.util.general, 247
pyrolite.util.lambdas, 273
pyrolite.util.lambdas.eval, 275
pyrolite.util.lambdas.oneill, 276
pyrolite.util.lambdas.opt, 277
pyrolite.util.lambdas.params, 274
pyrolite.util.lambdas.plot, 279
pyrolite.util.lambdas.transform, 280
pyrolite.util.log, 309
pyrolite.util.math, 269
pyrolite.util.meta, 290
pyrolite.util.missing, 289
pyrolite.util.pd, 249
pyrolite.util.plot, 251
pyrolite.util.plot.axes, 251
pyrolite.util.plot.density, 254
pyrolite.util.plot.export, 256
pyrolite.util.plot.grid, 257
pyrolite.util.plot.helpers, 258
pyrolite.util.plot.interpolation, 261
pyrolite.util.plot.legend, 262
pyrolite.util.plot.style, 263
pyrolite.util.plot.transform, 264
pyrolite.util.resampling, 286
pyrolite.util.skl, 291
pyrolite.util.skl.impute, 297
pyrolite.util.skl.pipeline, 293
pyrolite.util.skl.select, 295
pyrolite.util.skl.transform, 296
pyrolite.util.skl.vis, 291
pyrolite.util.spatial, 284
pyrolite.util.synthetic, 282
pyrolite.util.text, 265
pyrolite.util.time, 267
pyrolite.util.types, 290
pyrolite.util.units, 289
pyrolite.util.web, 267

INDEX

A

ABC_to_xy() (*in module pyrolite.util.plot.transform*), 265
accumulate() (*in module pyrolite.util.pd*), 249
add_age_noise() (*in module pyrolite.util.resampling*), 287
add_colorbar() (*in module pyrolite.util.plot.axes*), 253
add_endmember() (*pyrolite.mineral.template.Mineral method*), 239
add_MgNo() (*in module pyrolite.geochem.transform*), 218
add_MgNo() (*pyrolite.geochem.pyrochem method*), 209
add_ratio() (*pyrolite.geochem.pyrochem method*), 209
add_to_axes() (*pyrolite.util.classification.FeldsparTernary method*), 302
add_to_axes() (*pyrolite.util.classification.Herron method*), 307
add_to_axes() (*pyrolite.util.classification.JensenPlot method*), 303
add_to_axes() (*pyrolite.util.classification.Magnetite method*), 309
add_to_axes() (*pyrolite.util.classification.Pettijohn method*), 306
add_to_axes() (*pyrolite.util.classification.PolygonClassifier method*), 298
add_to_axes() (*pyrolite.util.classification.QAP method*), 301
add_to_axes() (*pyrolite.util.classification.SpinelFeBivariate method*), 305
add_to_axes() (*pyrolite.util.classification.SpinelTrivalentTernary method*), 304
add_to_axes() (*pyrolite.util.classification.TAS method*), 299
add_to_axes() (*pyrolite.util.classification.USDASoilTexture method*), 300
affine_transform() (*in module pyrolite.util.plot.transform*), 264
age_name() (*in module pyrolite.util.time*), 267
aggregate_element() (*in module pyrolite.geochem.transform*), 217
aggregate_element() (*pyrolite.geochem.pyrochem method*), 207
all_reference_compositions() (*in module pyrolite.geochem.norm*), 221
alphalabel_subplots() (*in module pyrolite.util.plot.helpers*), 258
alphas_from_multiclass_prob() (*in module pyrolite.util.skl.vis*), 292
ALR() (*in module pyrolite.comp.codata*), 231
ALR() (*pyrolite.comp.pyrocomp method*), 228
ALRTransform (*class in pyrolite.util.skl.transform*), 296
apfu() (*pyrolite.mineral.template.Mineral method*), 239
augmented_covariance_matrix() (*in module pyrolite.util.math*), 269
AX (*class in pyrolite.mineral.sites*), 243
axes_to_ternary() (*in module pyrolite.util.plot.axes*), 252
axis_components (*pyrolite.util.classification.FeldsparTernary property*), 302
axis_components (*pyrolite.util.classification.Herron property*), 308
axis_components (*pyrolite.util.classification.JensenPlot property*), 304
axis_components (*pyrolite.util.classification.Magnetite property*), 308
axis_components (*pyrolite.util.classification.Pettijohn property*), 307
axis_components (*pyrolite.util.classification.PolygonClassifier property*), 298
axis_components (*pyrolite.util.classification.QAP property*), 301
axis_components (*pyrolite.util.classification.SpinelFeBivariate property*), 306
axis_components (*pyrolite.util.classification.SpinelTrivalentTernary*

property), 305
axis_components (pyrolite.util.classification.TAS property), 299
axis_components (pyrolite.util.classification.USDASoilTexture property), 300

B

bin_centres_to_edges() (in module pyrolite.util.plot.grid), 257
bin_edges_to_centres() (in module pyrolite.util.plot.grid), 257
boxcox() (in module pyrolite.comp.codata), 234
boxcox() (pyrolite.comp.pyrocomp method), 229
BoxCoxTransform (class in pyrolite.util.skl.transform), 297
buf (pyrolite.util.log.ToLogger attribute), 309
build() (pyrolite.util.time.Timescale method), 268
by_incompatibility() (in module pyrolite.geochem.ind), 214
by_number() (in module pyrolite.geochem.ind), 215

C

calc_lambdas() (in module pyrolite.util.lambdas), 273
calculate_grid() (pyrolite.plot.density.grid.DensityGrid method), 196
calculate_occupancy() (pyrolite.mineral.template.Mineral method), 240
CCPI() (in module pyrolite.geochem.alteration), 226
check_default_axes() (in module pyrolite.util.plot.axes), 252
check_empty() (in module pyrolite.util.plot.axes), 252
check_multiple_cation_inclusion() (in module pyrolite.geochem.parse), 223
check_multiple_cation_inclusion() (pyrolite.geochem.pyrochem method), 206
checkpoint() (pyrolite.util.general.Timewith method), 247
CIA() (in module pyrolite.geochem.alteration), 224
CIPW_norm() (in module pyrolite.mineral.normative), 242
CIW() (in module pyrolite.geochem.alteration), 225
classifier_performance_report() (in module pyrolite.util.skl.pipeline), 293
close() (in module pyrolite.comp.codata), 230
CLR() (in module pyrolite.comp.codata), 231
CLR() (pyrolite.comp.pyrocomp method), 228
CLRTransform (class in pyrolite.util.skl.transform), 296
color_ternary_polygons_by_centroid() (in module pyrolite.util.plot.style), 264
column_ordered_append() (in module pyrolite.util.pd), 249
ColumnSelector (class in pyrolite.util.skl.select), 295
common_elements() (in module pyrolite.geochem.ind), 212
common_oxides() (in module pyrolite.geochem.ind), 213
Composition (class in pyrolite.geochem.norm), 221
compositional (pyrolite.geochem.pyrochem property), 206
compositional_biplot() (in module pyrolite.plot.biplot), 199
compositional_cosine_distances() (in module pyrolite.comp.codata), 235
compositional_mean() (in module pyrolite.comp.aggregate), 236
compositional_SVD() (in module pyrolite.plot.biplot), 198
CompositionalSelector (class in pyrolite.util.skl.select), 295
concat_columns() (in module pyrolite.util.pd), 250
conditional_prob_density() (in module pyrolite.util.plot.density), 255
convert_chemistry() (in module pyrolite.geochem.transform), 220
convert_chemistry() (pyrolite.geochem.pyrochem method), 210
cooccurence() (pyrolite.plot.pyroplot method), 189
cooccurence_pattern() (in module pyrolite.util.missing), 289
copy() (pyrolite.mineral.template.MineralTemplate method), 239
copy_file() (in module pyrolite.util.general), 248
cross_ratios() (in module pyrolite.comp.aggregate), 237

D

denormalize_from() (pyrolite.geochem.pyrochem method), 211
density() (in module pyrolite.plot.density), 195
density() (pyrolite.plot.pyroplot method), 189
DensityGrid (class in pyrolite.plot.density.grid), 196
describe() (pyrolite.geochem.norm.Composition method), 221
devolatilise() (in module pyrolite.geochem.transform), 216
devolatilise() (pyrolite.geochem.pyrochem method), 207
Devolatilizer (class in pyrolite.util.skl.transform), 297
df_from_csvs() (in module pyrolite.util.pd), 250
download_file() (in module pyrolite.util.web), 267
draw_vector() (in module pyrolite.util.plot.helpers), 259
drop_where_all_empty() (in module pyrolite.util.pd), 249
DropBelowZero (class in pyrolite.util.skl.transform), 296

E

eigsorted() (*in module pyrolite.util.math*), 269
elapsed (*pyrolite.util.general.Timewith property*), 247
ElementAggregator (*class in pyrolite.util.skl.transform*), 297
elemental_sum() (*in module pyrolite.geochem.transform*), 217
elemental_sum() (*pyrolite.geochem.pyrochem method*), 207
elements (*pyrolite.geochem.pyrochem property*), 205
ElementSelector (*class in pyrolite.util.skl.select*), 295
EMCOMP() (*in module pyrolite.comp.impute*), 237
endmember_decompose() (*in module pyrolite.mineral.normative*), 240
endmember_decompose() (*pyrolite.mineral.template.Mineral method*), 239
equal_within_significance() (*in module pyrolite.util.math*), 272
example_patterns_from_parameters() (*in module pyrolite.util.synthetic*), 284
example_spider_data() (*in module pyrolite.util.synthetic*), 284
ExpTransform (*class in pyrolite.util.skl.transform*), 296
extent_from_xy() (*pyrolite.plot.density.grid.DensityGrid method*), 196

F

FeAt8MgO() (*in module pyrolite.geochem.magma*), 223
FeldsparTernary (*class in pyrolite.util.classification*), 302
FeldsparTernary() (*in module pyrolite.plot.templates*), 200
fit() (*pyrolite.util.skl.pipeline.PdUnion method*), 295
fit() (*pyrolite.util.skl.select.ColumnSelector method*), 295
fit() (*pyrolite.util.skl.select.CompositionalSelector method*), 295
fit() (*pyrolite.util.skl.select.ElementSelector method*), 295
fit() (*pyrolite.util.skl.select.MajorsSelector method*), 295
fit() (*pyrolite.util.skl.select.REESelector method*), 296
fit() (*pyrolite.util.skl.select.TypeSelector method*), 295
fit() (*pyrolite.util.skl.transform.ALRTransform method*), 296
fit() (*pyrolite.util.skl.transform.BoxCoxTransform method*), 297
fit() (*pyrolite.util.skl.transform.CLRTransform method*), 296
fit() (*pyrolite.util.skl.transform.Devolatilizer method*), 297
fit() (*pyrolite.util.skl.transform.DropBelowZero method*), 296
fit() (*pyrolite.util.skl.transform.ElementAggregator method*), 297
fit() (*pyrolite.util.skl.transform.ExpTransform method*), 296
fit() (*pyrolite.util.skl.transform.ILRTransform method*), 297
fit() (*pyrolite.util.skl.transform.LambdaTransformer method*), 297
fit() (*pyrolite.util.skl.transform.LinearTransform method*), 296
fit() (*pyrolite.util.skl.transform.LogTransform method*), 296
fit() (*pyrolite.util.skl.transform.SphericalCoordTransform method*), 297
fit_lattice_strain() (*in module pyrolite.mineral.lattice*), 245
fit_save_classifier() (*in module pyrolite.util.skl.pipeline*), 293
flatten_dict() (*in module pyrolite.util.general*), 248
flattengrid() (*in module pyrolite.util.math*), 270
flush() (*pyrolite.util.log.ToLogger method*), 310
formula_to_elemental() (*in module pyrolite.mineral.transform*), 243

G

get_ALR_labels() (*in module pyrolite.comp.codata*), 233
get_axes_index() (*in module pyrolite.util.plot.axes*), 251
get_axis_density_methods() (*in module pyrolite.util.plot.density*), 254
get_cations() (*in module pyrolite.geochem.ind*), 214
get_centre_grid() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_centroid() (*in module pyrolite.util.plot.helpers*), 258
get_CLR_labels() (*in module pyrolite.comp.codata*), 233
get_cmode() (*in module pyrolite.plot.color*), 203
get_contour_paths() (*in module pyrolite.util.plot.interpolation*), 262
get_edge_grid() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_extent() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_full_column() (*in module pyrolite.comp.aggregate*), 235
get_full_extent() (*in module pyrolite.util.plot.export*), 256
get_function_components() (*in module pyrolite.util.lambdas.eval*), 276

get_hex_extent() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_ILR_labels() (*in module pyrolite.comp.codata*), 233
get_ionic_radii() (*in module pyrolite.geochem.ind*), 215
get_isotopes() (*in module pyrolite.geochem.ind*), 214
get_lambda_poly_function() (*in module pyrolite.util.lambdas.eval*), 276
get_mineral() (*in module pyrolite.mineral.mindb*), 246
get_mineral_group() (*in module pyrolite.mineral.mindb*), 247
get_module_datafolder() (*in module pyrolite.util.meta*), 290
get_ordered_axes() (*in module pyrolite.util.plot.axes*), 251
get_polynomial_matrix() (*in module pyrolite.util.lambdas.oneill*), 276
get_range() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_ratio() (*in module pyrolite.geochem.transform*), 218
get_ratio() (*pyrolite.geochem.pyrochem method*), 208
get_reference_composition() (*in module pyrolite.geochem.norm*), 221
get_reference_files() (*in module pyrolite.geochem.norm*), 221
get_scaler() (*in module pyrolite.util.distributions*), 281
get_site_occupancy() (*pyrolite.mineral.template.Mineral method*), 240
get_spatiotemporal_resampling_weights() (*in module pyrolite.util.resampling*), 286
get_tetrads_function() (*in module pyrolite.util.lambdas.eval*), 275
get_transforms() (*in module pyrolite.comp.codata*), 235
get_twins() (*in module pyrolite.util.plot.axes*), 253
get_visual_center() (*in module pyrolite.util.plot.helpers*), 258
get_xrange() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_xstep() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_yrange() (*pyrolite.plot.density.grid.DensityGrid method*), 196
get_ystep() (*pyrolite.plot.density.grid.DensityGrid method*), 196
great_circle_distance() (*in module pyrolite.util.spatial*), 284
grid_from_ranges() (*in module pyrolite.util.math*), 269

H

Handle() (*in module pyrolite.util.log*), 309
have_internet_connection() (*in module pyrolite.util.web*), 267
heatscatter() (*pyrolite.plot.pyroplot method*), 190
helmert_basis() (*in module pyrolite.util.math*), 272
Herron (*class in pyrolite.util.classification*), 307
Herron() (*in module pyrolite.plot.templates*), 200

I

ILR() (*in module pyrolite.comp.codata*), 232
ILR() (*pyrolite.comp.pyrocomp method*), 228
ILRTransform (*class in pyrolite.util.skl.transform*), 296
import_colors() (*in module pyrolite.util.time*), 268
inargs() (*in module pyrolite.util.meta*), 290
init_axes() (*in module pyrolite.util.plot.axes*), 252
init_spherical_octant() (*in module pyrolite.util.plot.helpers*), 260
int_to_alpha() (*in module pyrolite.util.text*), 267
interpolate_line() (*in module pyrolite.util.math*), 269
interpolate_path() (*in module pyrolite.util.plot.interpolation*), 261
interpolated_patch_path() (*in module pyrolite.util.plot.interpolation*), 261
inverse_ALR() (*in module pyrolite.comp.codata*), 231
inverse_ALR() (*pyrolite.comp.pyrocomp method*), 228
inverse_boxcox() (*in module pyrolite.comp.codata*), 234
inverse_boxcox() (*pyrolite.comp.pyrocomp method*), 229
inverse_CLR() (*in module pyrolite.comp.codata*), 232
inverse_CLR() (*pyrolite.comp.pyrocomp method*), 228
inverse_ILR() (*in module pyrolite.comp.codata*), 232
inverse_ILR() (*pyrolite.comp.pyrocomp method*), 229
inverse_sphere() (*in module pyrolite.comp.codata*), 235
inverse_sphere() (*pyrolite.comp.pyrocomp method*), 229
inverse_transform() (*pyrolite.util.skl.transform.ALRTransform method*), 296
inverse_transform() (*pyrolite.util.skl.transform.BoxCoxTransform method*), 297
inverse_transform() (*pyrolite.util.skl.transform.CLRTransform method*), 296
inverse_transform() (*pyrolite.util.skl.transform.ExpTransform method*), 296
inverse_transform() (*pyrolite.util.skl.transform.ILRTransform method*), 297

inverse_transform() (*pyrolite.util.skl.transform.LinearTransform method*), 296
inverse_transform() (*pyrolite.util.skl.transform.LogTransform method*), 296
inverse_transform() (*pyrolite.util.skl.transform.SphericalCoordTransform method*), 297
invert_transform() (*pyrolite.comp.pyrocomp method*), 230
is_isotoperatio() (*in module pyrolite.geochem.parse*), 222
is_numeric() (*in module pyrolite.util.math*), 271
ischem() (*in module pyrolite.geochem.parse*), 222
isclose() (*in module pyrolite.util.math*), 271
iscollection() (*in module pyrolite.util.types*), 290
IshikawaAltIndex() (*in module pyrolite.geochem.alteration*), 226
isotope_ratios (*pyrolite.geochem.pyrochem property*), 205
IX (*class in pyrolite.mineral.sites*), 243

J

JensenPlot (*class in pyrolite.util.classification*), 303
JensenPlot() (*in module pyrolite.plot.templates*), 201

K

kdefrom() (*pyrolite.plot.density.grid.DensityGrid method*), 196

L

label_axes() (*in module pyrolite.util.plot.axes*), 252
lambda_lnREE() (*in module pyrolite.geochem.transform*), 219
lambda_lnREE() (*pyrolite.geochem.pyrochem method*), 209
lambda_poly() (*in module pyrolite.util.lambdas.eval*), 275
lambdas_ONeill2016() (*in module pyrolite.util.lambdas.oneill*), 276
lambdas_optimize() (*in module pyrolite.util.lambdas.opt*), 278
LambdaTransformer (*class in pyrolite.util.skl.transform*), 297
LeMaitre_Fe_correction() (*in module pyrolite.mineral.normative*), 242
LeMaitreOxRatio() (*in module pyrolite.mineral.normative*), 241
level (*pyrolite.util.log.ToLogger attribute*), 310
levenshtein_distance() (*in module pyrolite.util.spatial*), 286
linear_fit_components() (*in module pyrolite.util.lambdas.opt*), 277
LinearTransform (*class in pyrolite.util.skl.transform*), 296
linekwargs() (*in module pyrolite.util.plot.style*), 263
linrng_() (*in module pyrolite.util.math*), 271
linspc_() (*in module pyrolite.util.math*), 270
list_compositional (*pyrolite.geochem.pyrochem property*), 205
list_elements (*pyrolite.geochem.pyrochem property*), 204
list_formulae() (*in module pyrolite.mineral.mindb*), 246
list_groups() (*in module pyrolite.mineral.mindb*), 246
list_isotope_ratios (*pyrolite.geochem.pyrochem property*), 204
list_minerals() (*in module pyrolite.mineral.mindb*), 246
list_oxides (*pyrolite.geochem.pyrochem property*), 205
list_REE (*pyrolite.geochem.pyrochem property*), 205
list_REY (*pyrolite.geochem.pyrochem property*), 205
listify() (*in module pyrolite.util.time*), 267
logger (*pyrolite.util.log.ToLogger attribute*), 310
lognorm_to_norm() (*in module pyrolite.util.distributions*), 281
logratiomean() (*in module pyrolite.comp.codata*), 232
logratiomean() (*pyrolite.comp.pyrocomp method*), 230
logrng_() (*in module pyrolite.util.math*), 270
logspc_() (*in module pyrolite.util.math*), 270
LogTransform (*class in pyrolite.util.skl.transform*), 296

M

Magnetite (*class in pyrolite.util.classification*), 308
Magnetite() (*in module pyrolite.plot.templates*), 203
MajorsSelector (*class in pyrolite.util.skl.select*), 295
mappable_from_values() (*in module pyrolite.util.plot.style*), 263
marker_cycle() (*in module pyrolite.util.plot.style*), 263
md_pattern() (*in module pyrolite.util.missing*), 289
merge_formulae() (*in module pyrolite.mineral.transform*), 243
Middlemost_Fe_correction() (*in module pyrolite.mineral.normative*), 241
MiddlemostOxRatio() (*in module pyrolite.mineral.normative*), 241
Mineral (*class in pyrolite.mineral.template*), 239
MineralTemplate (*class in pyrolite.mineral.template*), 238
models (*pyrolite.util.classification.Magnetite attribute*), 308
modify_legend_handles() (*in module pyrolite.util.plot.legend*), 262
module
    pyrolite.comp, 227
    pyrolite.comp.aggregate, 235

pyrolite.comp.codata, 230
pyrolite.comp.impute, 237
pyrolite.geochem, 204
pyrolite.geochem.alteration, 224
pyrolite.geochem.ind, 212
pyrolite.geochem.ions, 226
pyrolite.geochem.isotope, 227
pyrolite.geochem.magma, 223
pyrolite.geochem.norm, 221
pyrolite.geochem.parse, 222
pyrolite.geochem.transform, 216
pyrolite.mineral, 238
pyrolite.mineral.lattice, 244
pyrolite.mineral.mindb, 246
pyrolite.mineral.normative, 240
pyrolite.mineral.sites, 243
pyrolite.mineral.template, 238
pyrolite.mineral.transform, 243
pyrolite.plot, 189
pyrolite.plot.biplot, 198
pyrolite.plot.color, 203
pyrolite.plot.density, 195
pyrolite.plot.density.grid, 196
pyrolite.plot.density.ternary, 196
pyrolite.plot.parallel, 197
pyrolite.plot.spider, 193
pyrolite.plot.stem, 198
pyrolite.plot.templates, 199
pyrolite.util, 247
pyrolite.util.classification, 297
pyrolite.util.distributions, 281
pyrolite.util.general, 247
pyrolite.util.lambdas, 273
pyrolite.util.lambdas.eval, 275
pyrolite.util.lambdas.oneill, 276
pyrolite.util.lambdas.opt, 277
pyrolite.util.lambdas.params, 274
pyrolite.util.lambdas.plot, 279
pyrolite.util.lambdas.transform, 280
pyrolite.util.log, 309
pyrolite.util.math, 269
pyrolite.util.meta, 290
pyrolite.util.missing, 289
pyrolite.util.pd, 249
pyrolite.util.plot, 251
pyrolite.util.plot.axes, 251
pyrolite.util.plot.density, 254
pyrolite.util.plot.export, 256
pyrolite.util.plot.grid, 257
pyrolite.util.plot.helpers, 258
pyrolite.util.plot.interpolation, 261
pyrolite.util.plot.legend, 262
pyrolite.util.plot.style, 263
pyrolite.util.plot.transform, 264
pyrolite.util.resampling, 286
pyrolite.util.skl, 291
pyrolite.util.skl.impute, 297
pyrolite.util.skl.pipeline, 293
pyrolite.util.skl.select, 295
pyrolite.util.skl.transform, 296
pyrolite.util.skl.vis, 291
pyrolite.util.spatial, 284
pyrolite.util.synthetic, 282
pyrolite.util.text, 265
pyrolite.util.time, 267
pyrolite.util.types, 290
pyrolite.util.units, 289
pyrolite.util.web, 267
most_precise() (*in module pyrolite.util.math*), 272
MX (*class in pyrolite.mineral.sites*), 243

N

NaAt8MgO() (*in module pyrolite.geochem.magma*), 223
named_age() (*pyrolite.util.time.Timescale method*), 268
nan_scatter() (*in module pyrolite.util.plot.helpers*), 260
nan_weighted_compositional_mean() (*in module pyrolite.comp.aggregate*), 236
nan_weighted_mean() (*in module pyrolite.comp.aggregate*), 236
nancov() (*in module pyrolite.util.math*), 273
norm_to_lognorm() (*in module pyrolite.util.distributions*), 282
normal_frame() (*in module pyrolite.util.synthetic*), 283
normal_series() (*in module pyrolite.util.synthetic*), 284
normalise_whitespace() (*in module pyrolite.util.text*), 265
normalize_to() (*pyrolite.geochem.pyrochem method*), 211
np_cross_ratios() (*in module pyrolite.comp.aggregate*), 237
NSEW_2_bounds() (*in module pyrolite.util.spatial*), 285

O

on_finite() (*in module pyrolite.util.math*), 273
optimize_fit_components() (*in module pyrolite.util.lambdas.opt*), 278
orthogonal_polynomial_constants() (*in module pyrolite.util.lambdas.params*), 274
outliers() (*in module pyrolite.util.pd*), 250
OX (*class in pyrolite.mineral.sites*), 243
oxide_conversion() (*in module pyrolite.geochem.transform*), 217
oxides (*pyrolite.geochem.pyrochem property*), 205

P

parallel() (*in module pyrolite.plot.parallel*), 197

parallel() (*pyrolite.plot.pyroplot method*), 190
parse_chem() (*pyrolite.geochem.pyrochem method*), 206
parse_composition() (*in module pyrolite.mineral.mindb*), 247
parse_entry() (*in module pyrolite.util.text*), 266
parse_sigmas() (*in module pyrolite.util.lambdas.params*), 275
patchkwargs() (*in module pyrolite.util.plot.style*), 263
path_to_csv() (*in module pyrolite.util.plot.export*), 257
pcov_from_jac() (*in module pyrolite.util.lambdas.opt*), 277
PdUnion (*class in pyrolite.util.skl.pipeline*), 295
pearceThNbYb() (*in module pyrolite.plot.templates*), 202
pearceTiNbYb() (*in module pyrolite.plot.templates*), 202
PeralkalinityClassifier (*class in pyrolite.util.classification*), 303
percentile_contour_values_from_meshz() (*in module pyrolite.util.plot.density*), 254
Pettijohn (*class in pyrolite.util.classification*), 306
Pettijohn() (*in module pyrolite.plot.templates*), 201
PIA() (*in module pyrolite.geochem.alteration*), 225
piecewise() (*in module pyrolite.util.spatial*), 285
plot() (*pyrolite.plot.pyroplot method*), 191
plot_2dhull() (*in module pyrolite.util.plot.helpers*), 259
plot_confusion_matrix() (*in module pyrolite.util.skl.vis*), 291
plot_cooccurence() (*in module pyrolite.util.plot.helpers*), 259
plot_gs_results() (*in module pyrolite.util.skl.vis*), 292
plot_lambdas_components() (*in module pyrolite.util.lambdas.plot*), 279
plot_mapping() (*in module pyrolite.util.skl.vis*), 292
plot_origin_to_points() (*in module pyrolite.plot.biplot*), 198
plot_pca_vectors() (*in module pyrolite.util.plot.helpers*), 260
plot_profiles() (*in module pyrolite.util.lambdas.plot*), 280
plot_stdev_ellipses() (*in module pyrolite.util.plot.helpers*), 259
plot_tetrads_components() (*in module pyrolite.util.lambdas.plot*), 279
plot_Z_percentiles() (*in module pyrolite.util.plot.density*), 254
PolygonClassifier (*class in pyrolite.util.classification*), 297
predict() (*pyrolite.util.classification.FeldsparTernary method*), 303
predict() (*pyrolite.util.classification.Herron method*), 308
predict() (*pyrolite.util.classification.JensenPlot method*), 304
predict() (*pyrolite.util.classification.Magnetite method*), 308
predict() (*pyrolite.util.classification.PeralkalinityClassifier method*), 303
predict() (*pyrolite.util.classification.Pettijohn method*), 307
predict() (*pyrolite.util.classification.PolygonClassifier method*), 298
predict() (*pyrolite.util.classification.QAP method*), 302
predict() (*pyrolite.util.classification.SpinelFeBivariate method*), 306
predict() (*pyrolite.util.classification.SpinelTrivalentTernary method*), 305
predict() (*pyrolite.util.classification.TAS method*), 300
predict() (*pyrolite.util.classification.USDASoilTexture method*), 301
process_color() (*in module pyrolite.plot.color*), 203
proxy_line() (*in module pyrolite.util.plot.legend*), 262
proxy_rect() (*in module pyrolite.util.plot.legend*), 262
pyrochem (*class in pyrolite.geochem*), 204
pyrocomp (*class in pyrolite.comp*), 227
pyrolite.comp
    module, 227
pyrolite.comp.aggregate
    module, 235
pyrolite.comp.codata
    module, 230
pyrolite.comp.impute
    module, 237
pyrolite.geochem
    module, 204
pyrolite.geochem.alteration
    module, 224
pyrolite.geochem.ind
    module, 212
pyrolite.geochem.ions
    module, 226
pyrolite.geochem.isotope
    module, 227
pyrolite.geochem.magma
    module, 223
pyrolite.geochem.norm
    module, 221
pyrolite.geochem.parse
    module, 222
pyrolite.geochem.transform
    module, 216
pyrolite.mineral
    module, 238
pyrolite.mineral.lattice

module, 244
pyrolite.mineral.mindb
module, 246
pyrolite.mineral.normative
module, 240
pyrolite.mineral.sites
module, 243
pyrolite.mineral.template
module, 238
pyrolite.mineral.transform
module, 243
pyrolite.plot
module, 189
pyrolite.plot.biplot
module, 198
pyrolite.plot.color
module, 203
pyrolite.plot.density
module, 195
pyrolite.plot.density.grid
module, 196
pyrolite.plot.density.ternary
module, 196
pyrolite.plot.parallel
module, 197
pyrolite.plot.spider
module, 193
pyrolite.plot.stem
module, 198
pyrolite.plot.templates
module, 199
pyrolite.util
module, 247
pyrolite.util.classification
module, 297
pyrolite.util.distributions
module, 281
pyrolite.util.general
module, 247
pyrolite.util.lambdas
module, 273
pyrolite.util.lambdas.eval
module, 275
pyrolite.util.lambdas.oneill
module, 276
pyrolite.util.lambdas.opt
module, 277
pyrolite.util.lambdas.params
module, 274
pyrolite.util.lambdas.plot
module, 279
pyrolite.util.lambdas.transform
module, 280
pyrolite.util.log
module, 309
pyrolite.util.math
module, 269
pyrolite.util.meta
module, 290
pyrolite.util.missing
module, 289
pyrolite.util.pd
module, 249
pyrolite.util.plot
module, 251
pyrolite.util.plot.axes
module, 251
pyrolite.util.plot.density
module, 254
pyrolite.util.plot.export
module, 256
pyrolite.util.plot.grid
module, 257
pyrolite.util.plot.helpers
module, 258
pyrolite.util.plot.interpolation
module, 261
pyrolite.util.plot.legend
module, 262
pyrolite.util.plot.style
module, 263
pyrolite.util.plot.transform
module, 264
pyrolite.util.resampling
module, 286
pyrolite.util.skl
module, 291
pyrolite.util.skl.impute
module, 297
pyrolite.util.skl.pipeline
module, 293
pyrolite.util.skl.select
module, 295
pyrolite.util.skl.transform
module, 296
pyrolite.util.skl.vis
module, 291
pyrolite.util.spatial
module, 284
pyrolite.util.synthetic
module, 282
pyrolite.util.text
module, 265
pyrolite.util.time
module, 267
pyrolite.util.types
module, 290
pyrolite.util.units

module, 289
pyrolite.util.web
module, 267
pyrolite_datafolder() (*in module pyrolite.util.meta*), 290
pyroplot (*class in pyrolite.plot*), 189

Q

QAP (*class in pyrolite.util.classification*), 301
QAP() (*in module pyrolite.plot.templates*), 199
quoted_string() (*in module pyrolite.util.text*), 265

R

random_composition() (*in module pyrolite.util.synthetic*), 282
random_cov_matrix() (*in module pyrolite.util.synthetic*), 282
read_table() (*in module pyrolite.util.pd*), 249
recalc_cations() (*in module pyrolite.mineral.transform*), 243
recalculate_cations() (*pyrolite.mineral.template.Mineral method*), 239
recalculate_Fe() (*pyrolite.geochem.pyrochem method*), 208
rect_from_centre() (*in module pyrolite.util.plot.helpers*), 259
REE (*pyrolite.geochem.pyrochem property*), 205
REE() (*in module pyrolite.geochem.ind*), 212
REE() (*pyrolite.plot.pyroplot method*), 191
REE_radii_to_z() (*in module pyrolite.util.lambdas.transform*), 280
REE_v_radii() (*in module pyrolite.plot.spider*), 194
REE_z_to_radii() (*in module pyrolite.util.lambdas.transform*), 280
REESelector (*class in pyrolite.util.skl.select*), 296
remove_directory() (*in module pyrolite.util.general*), 248
remove_prefix() (*in module pyrolite.util.text*), 265
remove_suffix() (*in module pyrolite.util.text*), 265
renormalise() (*in module pyrolite.comp.codata*), 231
renormalise() (*pyrolite.comp.pyrocomp method*), 227
replace_with_ternary_axis() (*in module pyrolite.util.plot.axes*), 251
repr_isotope_ratio() (*in module pyrolite.geochem.parse*), 222
REY (*pyrolite.geochem.pyrochem property*), 205
REY() (*in module pyrolite.geochem.ind*), 213

S

sample_kde() (*in module pyrolite.util.distributions*), 281
sample_ternary_kde() (*in module pyrolite.util.distributions*), 281
SAR() (*in module pyrolite.geochem.alteration*), 225
save_axes() (*in module pyrolite.util.plot.export*), 256
save_figure() (*in module pyrolite.util.plot.export*), 256
scale() (*in module pyrolite.util.units*), 289
scale() (*pyrolite.geochem.pyrochem method*), 211
scatter() (*pyrolite.plot.pyroplot method*), 191
scatterkwargs() (*in module pyrolite.util.plot.style*), 263
SCSS() (*in module pyrolite.geochem.magma*), 224
set_composition() (*pyrolite.mineral.template.Mineral method*), 239
set_DB() (*in module pyrolite.geochem.norm*), 221
set_default_ionic_charges() (*in module pyrolite.geochem.ions*), 226
set_endmembers() (*pyrolite.mineral.template.Mineral method*), 239
set_structure() (*pyrolite.mineral.template.MineralTemplate method*), 238
set_template() (*pyrolite.mineral.template.Mineral method*), 239
set_units() (*pyrolite.geochem.norm.Composition method*), 221
share_axes() (*in module pyrolite.util.plot.axes*), 253
significant_figures() (*in module pyrolite.util.math*), 271
signify_digit() (*in module pyrolite.util.math*), 271
simple_oxides() (*in module pyrolite.geochem.ind*), 213
Site (*class in pyrolite.mineral.sites*), 243
SiTiIndex() (*in module pyrolite.geochem.alteration*), 225
slugify() (*in module pyrolite.util.text*), 266
solve_ratios() (*in module pyrolite.util.math*), 273
spatiotemporal_bootstrap_resample() (*in module pyrolite.util.resampling*), 287
spatiotemporal_split() (*in module pyrolite.util.spatial*), 285
sphere() (*in module pyrolite.comp.codata*), 234
sphere() (*pyrolite.comp.pyrocomp method*), 229
SphericalCoordTransform (*class in pyrolite.util.skl.transform*), 297
sphinx_doi_link() (*in module pyrolite.util.meta*), 290
spider() (*in module pyrolite.plot.spider*), 193
spider() (*pyrolite.plot.pyroplot method*), 192
SpinelFeBivariate (*class in pyrolite.util.classification*), 305
SpinelFeBivariate() (*in module pyrolite.plot.templates*), 201
SpinelTrivalentTernary (*class in pyrolite.util.classification*), 304
SpinelTrivalentTernary() (*in module pyrolite.plot.templates*), 202
split_records() (*in module pyrolite.util.text*), 266

standardise_aggregate() (*in module pyrolite.comp.aggregate*), 237
stem() (*in module pyrolite.plot.stem*), 198
stem() (*pyrolite.plot.pyroplot method*), 192
strain_coefficient() (*in module pyrolite.mineral.lattice*), 244
string_variations() (*in module pyrolite.util.text*), 266
subaxes() (*in module pyrolite.util.plot.axes*), 253
subkwargs() (*in module pyrolite.util.meta*), 290
SVC_pipeline() (*in module pyrolite.util.skl.pipeline*), 294
swap_item() (*in module pyrolite.util.general*), 248
symbolic_helmert_basis() (*in module pyrolite.util.math*), 272

T

take_me_to_the_docs() (*in module pyrolite.util.meta*), 290
TAS (*class in pyrolite.util.classification*), 299
TAS() (*in module pyrolite.plot.templates*), 199
temp_path() (*in module pyrolite.util.general*), 248
tempdir() (*in module pyrolite.util.general*), 248
ternary_color() (*in module pyrolite.util.plot.style*), 263
ternary_grid() (*in module pyrolite.util.plot.grid*), 257
ternary_heatmap() (*in module pyrolite.plot.density.ternary*), 196
tetrad() (*in module pyrolite.util.lambdas.eval*), 275
text2age() (*pyrolite.util.time.Timescale method*), 268
Timescale (*class in pyrolite.util.time*), 268
timescale_reference_frame() (*in module pyrolite.util.time*), 268
Timewith (*class in pyrolite.util.general*), 247
titlecase() (*in module pyrolite.util.text*), 265
tlr_to_xy() (*in module pyrolite.util.plot.transform*), 264
to_frame() (*in module pyrolite.util.pd*), 249
to_molecular() (*in module pyrolite.geochem.transform*), 216
to_molecular() (*pyrolite.geochem.pyrochem method*), 206
to_numeric() (*in module pyrolite.util.pd*), 250
to_ser() (*in module pyrolite.util.pd*), 249
to_weight() (*in module pyrolite.geochem.transform*), 216
to_weight() (*pyrolite.geochem.pyrochem method*), 206
to_width() (*in module pyrolite.util.text*), 265
tochem() (*in module pyrolite.geochem.parse*), 222
ToLogger (*class in pyrolite.util.log*), 309
transform() (*pyrolite.util.skl.pipeline.PdUnion method*), 295
transform() (*pyrolite.util.skl.select.ColumnSelector method*), 295
transform() (*pyrolite.util.skl.select.CompositionalSelector method*), 295
transform() (*pyrolite.util.skl.select.ElementSelector method*), 296
transform() (*pyrolite.util.skl.select.MajorsSelector method*), 295
transform() (*pyrolite.util.skl.select.REESelector method*), 296
transform() (*pyrolite.util.skl.select.TypeSelector method*), 295
transform() (*pyrolite.util.skl.transform.ALRTransform method*), 296
transform() (*pyrolite.util.skl.transform.BoxCoxTransform method*), 297
transform() (*pyrolite.util.skl.transform.CLRTransform method*), 296
transform() (*pyrolite.util.skl.transform.Devolatilizer method*), 297
transform() (*pyrolite.util.skl.transform.DropBelowZero method*), 296
transform() (*pyrolite.util.skl.transform.ElementAggregator method*), 297
transform() (*pyrolite.util.skl.transform.ExpTransform method*), 296
transform() (*pyrolite.util.skl.transform.ILRTransform method*), 297
transform() (*pyrolite.util.skl.transform.LambdaTransformer method*), 297
transform() (*pyrolite.util.skl.transform.LinearTransform method*), 296
transform() (*pyrolite.util.skl.transform.LogTransform method*), 296
transform() (*pyrolite.util.skl.transform.SphericalCoordTransform method*), 297
TX (*class in pyrolite.mineral.sites*), 243
TypeSelector (*class in pyrolite.util.skl.select*), 295

U

uniques_from_concat() (*in module pyrolite.util.pd*), 250
univariate_distance_matrix() (*in module pyrolite.util.resampling*), 286
unmix() (*in module pyrolite.mineral.normative*), 240
update_database() (*in module pyrolite.geochem.norm*), 221
update_database() (*in module pyrolite.mineral.mindb*), 247
update_docstring_references() (*in module pyrolite.util.meta*), 291
update_grid_centre_ticks() (*pyrolite.plot.density.grid.DensityGrid method*), 196
update_grid_edge_ticks() (*pyrolite.plot.density.grid.DensityGrid method*),

196
urlify() (*in module pyrolite.util.web*), 267
USDASoilTexture (*class in pyrolite.util.classification*), 300
USDASoilTexture() (*in module pyrolite.plot.templates*), 202

V

vector_to_line() (*in module pyrolite.util.plot.helpers*), 259
VX (*class in pyrolite.mineral.sites*), 243

W

weights_from_array() (*in module pyrolite.comp.aggregate*), 235
WIP() (*in module pyrolite.geochem.alteration*), 226
write() (*pyrolite.util.log.ToLogger method*), 310

X

xy_to_ABC() (*in module pyrolite.util.plot.transform*), 265
xy_to_tlr() (*in module pyrolite.util.plot.transform*), 264

Y

youngs_modulus_approximation() (*in module pyrolite.mineral.lattice*), 245

Z

zero_to_nan() (*in module pyrolite.util.pd*), 250