iniabu

Release 1.1.2

Jul 13, 2023

Contents

CHAPTER 1

Introduction

The goal of this project is to give you access to various published solar abundance tables of elements as isotopes from a Python terminal. As one might guess, `iniabu` stands for initial abundances. Functions that are useful for everyday's life as an astrophysicist, cosmo-, or geochemist, are here made available in an easy to use interface. More information can also be found in the *scientific background section*. Aside from querying the databases, the `iniabu` tool allows you directly to calculate isotope ratios, δ-values, or ratios in bracket notation (the usual astronomy / logarithmic ratio notation) for fast comparison with your observations or measurements. These calculations can also be performed "numpy-style", i.e., element-wise on whole arrays.

The iniabu project is a young undertaking of the newly established Galactic Forensics Laboratory. We strive to support the open source culture and would be happy to hear how you use this tool, but also to see what you would like to have improved.

To get you started, check out the *Installation and Usage* page. If you would like to contribute, which we for sure welcome, please have a look at our, hopefully detailed, *Developers Guide*. We welcome all contributions, from simple typo fixes in the documentation, to feature contributions!

CHAPTER 2

Issues and feature requests

If you find a bug or have other problems or questions with this project, have a look at the GitHub issue page. Your issue might already have been discussed. Otherwise feel free to open a new issue. Please be as detailed as possible.

If you would like a new feature, please feel free to file a feature request. If you are interested in contributing this feature yourself, say so and we can help to get you started.

CHAPTER 3

Contributing

Contributions to this project are welcome. Please see the Developers Guide for detailed instructions on how to contribute.

CHAPTER 4

Relationship to previous package

This package is in its idea based on a software package that was was written at Lawrence Livermore National Laboratory, and replaces that specific package, which can in it's archived form be found here. That package's latest release was v0.3.1 and it was released under GPLv2. Note that the codebase for this version was rewritten from ground-up.

This new and improved version is available on PyPi and replaces the previous version there. The new package is not backwards compatible and is initially released as v1.0.0.

CHAPTER 5

Contents

5.1 Installation and Usage

5.1.1 Dependencies

This package is tested with python versions 3.7 - 3.10. It might work on older python version as well, however, compatibility is not guaranteed.

The only dependency at the moment for running the `iniabu` package is `numpy`. There is currently no pinned `numpy` version and the latest one should be working great.

5.1.2 Installation

To install the latest stable version of `iniabu`, run this command in your terminal:

```
$ pip install iniabu
```

Alternatively you can install `iniabu` directly from GitHub. This will install the latest version. To do so, type in your terminal:

```
$ pip install git+https://github.com/galactic-forensics/iniabu.git
```

5.1.3 Available databases

Several databases are available to work with. The current default database is called "lodders09" and is based on Lodders et al. (2009). This might be updated in the future without considering it a breaking change. Old databases will always stay available. Further databases, listed by the string used to call them, are as following:

- "asplund09": Asplund et al. (2009)
- "lodders09" (current default): Lodders et al. (2009)

- "nist": NIST database

The solar abundances of all databases were converted to number abundances and are relative to Si = 10^6. Conversion to other units, as described below, is possible.

Note: Not all databases mentioned here contain the solar abundances for every isotope. If an operation you are trying to perform encounters a solar abundance value that is not availagle in the currently loaded database, the result will be returned as an `np.nan`, i.e., as not a number.

5.1.4 Usage

Here we give a short overview of the `iniabu` module. Please also have a look at the *API Reference*. There, each module is described in detail, often with examples on how to use the specific function.

Furthermore, some examples in the form of Jupyter notebooks can be found on GitHub in docs/jupyter_examples.

Importing the module

Once installed, you can simply import the package from your python session as:

```
>>> from iniabu import ini
```

This is the recommended import and will be used throughout the rest of this documentation, unless otherwise noted. Here, the `ini` instance will be loaded with the default database (currently Lodders et al., 2009) and using linear, number abundances. Alternatively you can directly import the database using number, logarithmic abundances or mass fractions. The respective imports for these are:

```
>>> from iniabu import inilog  # number logarithmic abundances
>>> from iniabu import inimf  # mass fraction
```

Note: While the current default database is Lodders et al. (2009) we will not consider loading a different default database a breaking change. If you want to always load Lodders et al. (2009), use the specific import below.

In case multiple databases are required at the same time, e.g., `db1` using Lodders et al. (2009) and `db2` using Asplund et al. (2009) values and number logarithmic units, the following import could be used:

```
>>> import iniabu
>>> db1 = iniabu.IniAbu(database="lodders09")
>>> db2 = iniabu.IniAbu(database="asplund09", unit="num_log")
```

Loading a database

Switching the data base from a given instance `ini` can be easily accomplished. For example, the "asplund09" database can easily be loaded into a given instance by calling:

```
>>> ini.database = "asplund09"
```

Note: Switching a database does not reset the units. For example: If "lodders09" is loaded using mass fractions and you load "asplund09" as the new database, the units will stay the same that are used by default. A message will be printed to reflect this.

```
>>> ini.database = 'asplund09'
iniabu loaded database: 'asplund09', current units: 'mass_fraction'
```

Available abundance units

Abundance units can easily be switched between linear number abundances, logarithmic number abundances, and mass fraction units.

In the linear number abundances case all abundances are linear with respect to each other and are normalized such that the abundance of silicon is equal to 10^6 by number.

The logarithmic number abundances are generally used in astronomy. For an element X, the logarithmic abundance is defined with respect to the abundance of hydrogen as:

$$\log_{10}(\epsilon_X) = \log_{10}\left(\frac{\mathrm{N_X}}{\mathrm{N_H}}\right) + 12$$

Mass fraction values are common in nucleosynthesis calculations. To return mass fraction values the database can be switched to *mass_fraction*. The abundances are then defined as following:

$$X_i = \frac{N_i m_i}{\rho N_A}$$

Here X_i is the mass fraction of element i, N_i its number abundance, m_i its molecular mass, and N_N Avogadro's constant. The density ρ is defined as:

$$\rho = \frac{1}{N_A}\sum_i N_i m_i$$

To switch a given database between linear number abundance ("num_lin"), logarithmic number abundance ("num_log") mode, and mass fraction mode ("mass_fraction") the following property can be set:

```
>>> ini.unit == "num_log"
```

In this case, we would switch to logarithmic number abundance mode. To check what abundance unit is currently set, the following command can be used:

```
>>> ini.unit
"num_log"
```

By default, linear number abundance values are used.

Note: To use "num_log" or "mass_fraction" mode by default you can import the module in the following ways:

```
from iniabu import inilog  # "num_log" units
from iniabu import inimf  # "mass_fraction" units
```

Note: If you use "mass_fraction" units, the relative abundances of the isotopes are also given in mass fractions!

Element and isotope properties

Properties of an element are independent from the loaded database and are taken from the NIST database. To query the loaded database for relative or solar abundances, see the next two sections.

Querying an element:

To query an element's properties with respect to the solar abundance, it can be loaded into a temporary variable. For example: To query silicon the element and its properties can be loaded into a variable as following:

```
>>> ele = ini.ele["Si"]
```

Note that element names are not case sensitive. The following properties can now be queried from the element:

- The name of an element, which just returns that same abbreviation used to call the element, can be queried with `name`.
- The mass of the element, calculated using the isotope masses and the currently loaded abundances, using `mass`.
- The solar abundance of the element itself using `abu_solar`, normed as discussed above
- The mass number of its (stable) isotopes using `iso_a`
- The relative abundances of its (stable) isotopes using `iso_abu_rel`. If you are using "mass_fractions" as units, the relative abundances will also be given as mass fractions!
- The solar abundances of its (stable) isotopes using `iso_abu_solar`
- The number of protons of an element can be queried with `z`.

For example, to query the solar abundance of iron one could run the following statement:

```
>>> ele = ini.ele["Fe"]
>>> ele.abu_solar
847990.0
```

Note: You can query multiple elements as once. To do so, simply pass a list of the elements to be queried.

Querying an isotope

To query an isotope's properties with respect to the solar abundance, it can be loaded into a temporary variable, similar to when loading an element. For example: To query ^{54}Fe, the isotope can be loaded as a variable as following:

```
>>> iso = ini.iso["Fe-54"]
```

You can also use alternative spellings for the isotope name, e.g., `"54Fe"` or `"Fe54"`. Furthermore, none of these spellings are case sensitive. The following properties can then be queried from this isotope:

- The number of nucleons / mass number of an isotope can be queried with `a`.
- The name of the isotope(s) requested can be queried with `name`. These names will always be in the standard format, e.g., `"Fe-54"`.
- The mass of a specific isotope using `mass`.
- The solar abundance of the isotope itself using `abu_solar`, normed as discussed above
- The relative abundance of the specific isotope with respect to the element using `abu_rel`. *Note*: All isotopes of an element would sum up to a relative abundance of 1. If you are using "mass_fractions" as units, the relative abundances will also be given as mass fractions!
- The number of protons of an isotope can be queried with `z`.

Note: All isotope functions can be suffixed with *_all*. This will return information on all available isotopes, including unstable ones. Of course, solar system abundances for these are not available and will be returned as zeros, however, this might be useful to query masses.

For example: To query the solar and the relative abundances of ^{54}Fe one could run the following two commands in python:

```
>>> iso = ini.iso["Fe-54"]
>>> iso.abu_solar
49600.0
>>> iso.abu_rel
0.058449999999999995
```

Note: To query all isotopes of an element, you can query the isotope as following:

```
>>> iso = ini.iso["Ne"]
>>> iso.name
['Ne-20', 'Ne-21', 'Ne-22']
```

Note: You can query multiple isotopes at once. To do so, simply pass a list of the isotopes (or even elements in case of all isotopes) to be queried.

Element and isotope ratios

This function is used to calculate element and isotope ratios. Sure, the same can be accomplished by simply dividing the abundances of two isotopes. However, this function has some added benefits:

- Select if ratio is number fraction (default) or mass fraction
- Return multiple elements or isotopes at once

Some additional benefits when calculating isotope ratios:

- Choosing an element as the nominator selects all isotopes of the given element for the nominator
- Choosing an element as the denominator calculates the ratio for every isotope in the nominator with respect to the most abundant isotope of the element given as the denominator. This might sound complicated, but can be very useful since isotope ratios are often given with the most abundant isotope in the denominator

Note: If multiple isotope ratios are returned the function automatically returns them as a numpy array. This facilitates subsequent mathematical operations using these ratios.

The functions to calculate these ratios are called `ele_ratio` and `iso_ratio`. Below are some examples that describe some standard usage of these routines:

Caution: In these examples we assume that the database is loaded with "num_lin" units. If you are using "mass_fraction" units, you will get "mass_fraction" units back, even if you do not set `mass_fraction=True`. However, you could overwrite this behavior (the same way you can return *mass_fractions* even if you are in "num_lin" mode) by setting `mass_fraction=False`.

Some examples for elemental ratios:

- Calculate He to Pb ratio using number fraction and mass fraction: Here we assume that number, linear units are loaded:

```
>>> ini.ele_ratio("He", "Pb")  # number fraction
759537205.0816697
>>> ini.ele_ratio("He", "Pb", mass_fraction=True)
39321659726.58637
```

- Calculate multiple element ratios with the same denominator. The specific example here ratios Fe and Ni to Si:

```
>>> ini.ele_ratio(["Fe", "Ni"], "Si")
array([0.84824447, 0.04910773])
```

- Calculate multiple element ratios that have individual nominators and denominators. Here Si to Fe and Ni to Zr is calculated:

```
>>> ini.ele_ratio(["Si", "Ni"], ["Fe", "Zr"])
array([1.17890541e+00, 4.55450413e+03])
```

Some examples for isotope ratios:

- Calculate the isotope ratios of ^{6}Li to ^{7}Li as number fractions and as mass fractions. Here we assume that number, linear units are loaded:

```
>>> ini.iso_ratio("Li-6", "Li-7")  # number fractions by default
0.08212225817272835
>>> ini.iso_ratio("Li-6", "Li-7", mass_fraction=True)
0.09578691181324486
```

- Calculate isotope fractions of ^{3}He to ^{4}He and ^{21}Ne to ^{20}Ne:

```
>>> ini.iso_ratio(["He-3", "Ne-21"], ["He-4", "Ne-20"])
array([0.00016603, 0.00239717])
```

- Calculate the isotope ratios of all Si isotopes with respect to ^{28}Si. Three methods, all identical, are specified as following:
  - Method 1: The manual way specifying each isotope individually
  - Method 2: Select element in nominator chooses all isotopes of specified element
  - Method 3: The fastest way for this specific case is to choose *'Si'* as the element in the nominator and to choose *'Si'* in the denominator. The latter will pick the most abundant isotope of silicon, which is ^{28}Si.

```
>>> ini.iso_ratio(["Si-28", "Si-29", "Si-30"], "Si-28")  # Method 1
array([1.        , 0.05077524, 0.03347067])
>>> ini.iso_ratio("Si", "Si-28")  # Method 2
array([1.        , 0.05077524, 0.03347067])
>>> ini.iso_ratio("Si", "Si")  # Method 3
array([1.        , 0.05077524, 0.03347067])
```

δ-values

Note: A detailed discussion of δ-values can be found in the *Background Information*

The δ-value of a given isotope ratio, generally used in cosmo- and geochemistry, is defined as:

$$\delta\left(\frac{^iX}{^jX}\right) = \left(\frac{\left(\frac{^iX}{^jX}\right)_{\text{measured}}}{\left(\frac{^iX}{^jX}\right)_{\text{solar}}} - 1\right) \times f$$

Here the isotopes chosen for the ratio are ^iX and ^jX. The measured isotope ratio, which is in the nominator, is a value that must be provided to the function. The solar isotope ratio (denominator) will be taken from the solar abundance table using the isotope ratios provided to the routine. The factor f is by default set to 1000. This means that δ-values are by default returned as parts-per-thousand (‰). Choosing a different factor can be done by setting the keyword argument `delta_factor` accordingly.

Furthermore, the keyword argument `mass_fraction` can also be used as for ratios. Setting this keyword to `True` or `False` allows the user to overwrite the behavior of the loaded units.

While δ-values are commonly calculated for isotopes of one individual element, the routine allows to calculate δ-values between isotopes of different elements. To calculate a δ-values of two elements, the `ele_delta` function should be used. The equation given above represents a specific, but most commonly used case.

Finally: The `iso_delta` and `ele_delta` functions have the same features for specifying the nominator and denominator as the `iso_ratio` and `ele_ratio` functions mentioned above.

Caution: The values must be given in the same shape as the number of ratios provided. Otherwise the routine will return a `ValueError` specifying that there was a length mismatch.

Some examples for calculating δ-values for isotopes:

- Calculate one δ-value with a given measurement value. Here for ^{29}Si/^{28}Si. First calculated in parts per thousand (default), then as percent.

```
>>> ini.iso_delta("Si-30", "Si-28", 0.04)  # parts per thousand (default)
195.0761256883704
>>> ini.iso_delta("Si-30", "Si-28", 0.04, delta_factor=100)  # percent
19.50761256883704
```

- Calculate multiple δ-values as mass fractions. Here we calculate all Si isotopes with respect to ^{28}Si. Measurements are defined first. Three versions are provided that yield the same result. See description on calculating isotope ratios above for more detail.

```
>>> msr = [1., 0.01, 0.04]  # measurement
>>> ini.iso_delta(["Si-28", "Si-29", "Si-30"], "Si-28", msr)
array([   0.        , -803.05359812,  195.07612569])
>>> ini.iso_delta("Si", "Si-28", msr)
array([   0.        , -803.05359812,  195.07612569])
>>> ini.iso_delta("Si", "Si", msr)
array([   0.        , -803.05359812,  195.07612569])
```

- Calculate the δ-value for ^{84}Sr with respect to the major Sr isotope (^{86}Sr). The measurement value is provided as a mass fraction (assumption), but the database is loaded using number, linear units:

```
>>> ini.iso_delta("Sr-84", "Sr", 0.01, mass_fraction=True)
414.3962670607242
```

Some examples for calculating δ-values for elements:

- Calculate a δ-value for multiple elements, here Si and Ne with respect to Fe:

```
>>>  ini.ele_delta(["Si", "Ne"], "Fe", [2, 4])
array([696.48894668,  30.26124356])
```

Bracket-notation

The bracket notation, generally used in astronomy, for a given elemental ratio is defined as:

$$[\mathrm{X/Y}] = \log_{10}\left(\frac{N_\mathrm{X}}{N_\mathrm{Y}}\right)_\mathrm{star} - \log_{10}\left(\frac{N_\mathrm{X}}{N_\mathrm{Y}}\right)_\mathrm{solar}$$

Here, star stands for an arbitrary measurement, e.g., of a given star. X and Y are the elements of interest in this case, N_X and N_Y represent the respective number abundances of elements X and Y. Calculations with mass fractions are also allowed by the routine.

While bracket notation is commonly used with elements, there is no mathematical reason to prohibit using it for isotopes. Therefore, two routines are provided, namely `ele_bracket` and `iso_bracket`.

Finally: The `ele_bracket` and `iso_bracket` functions have the same features for specifying the nominator and denominator as the `iso_ratio` and `ele_ratio` functions mentioned above.

Some examples for calculating bracket notation values for elements:

- Calculate bracket notation value for Fe / H for a given measurement. First we calculate it as a number fraction (default setting) then as a mass fraction while having the database loaded in number linear mode.

```
>>> ini.ele_bracket("Fe", "H", 0.005)  # number fraction
2.183887471873783
>>> ini.ele_bracket("Fe", "H", 0.005, mass_fraction=True)  # mass fraction
3.9274378849968263
```

- Calculate bracket notation value for multiple measurements. Here, for O and Fe with respect to Fe.

```
>>> ini.ele_bracket(["O", "Fe"], "H", [0.02, 0.005])
array([1.51740521, 2.18388747])
```

Some examples for calculating bracket notation values for isotopes:

- Calculate a bracket notation values for multiple isotopes. Here for all Si isotopes with respect to ^{28}Si. *Note*: See `ratio_isotopes` for a detailed description of the possibilities.

```
>>> msr = [1., 0.01, 0.04]
>>>  ini.iso_bracket(["Si-28", "Si-29", "Si-30"], "Si-28", msr)
array([ 0.        , -0.70565195,  0.07739557])
>>> ini.iso_bracket("Si", "Si-28", msr)
array([ 0.        , -0.70565195,  0.07739557])
>>> ini.iso_bracket("Si", "Si", msr)
array([ 0.        , -0.70565195,  0.07739557])
```

Internal normalization

Internal normalization normalizes isotope ratios to two isotopes in order to remove any effects due to mass-dependent fractionation. A detailed explanation and further references can be found in the section *Background Information*.

Note: Internal normalization is only available for isotopes at this point. Elemental measurements generally suffer from effects other than mass-dependent fractionation. The math could of course be applied to elements as well,

however, can currently not be done with `iniabu`.

Several inputs are required for internal normalization. These are:

- The nominator isotope(s)
- The major and minor normalization isotopes
- The nominator isotope abundance(s) in the sample
- The normalization isotope abundances

The normalization isotopes and respective abundances must be given as a tuple or list with the main normalization isotope first. The minor normalization isotope (second) is the one used to correct mass-dependent fractionation.

You can also select the `delta_factor`. This is the multiplier by which the internally normalized value is multiplied at the end. By default, this factor is set to 10,000 and thus gives deviations in parts per 10,000. In geo- and cosmochemistry these deviations are often referred to as ϵ-values.

By default, an internally normalized value is calculated using the exponential law `law="exp"`. However, you can also choose to use the linear law by setting `law="lin"`.

Some examples:

- Normalize ^{60}Ni internally with respect to ^{58}Ni and ^{62}Ni. Use some made-up values for the data.

```
>>> ni58_counts = 1000000
>>> ni60_counts = 250000
>>> ni62_counts = 10000
>>> norm_counts = (ni58_counts, ni62_counts)
>>> ini.iso_int_norm("Ni-60", ("Ni-58", "Ni-62"), ni60_counts, norm_counts)
5145.864708640091
```

- Now this value is large to express in parts per 10,000. Let's switch the units to permil.

```
>>> ini.iso_int_norm("Ni-60", ("Ni-58", "Ni-62"), ni60_counts, norm_counts,
                     delta_factor=1000)
514.5864708640091
```

- If all nickel isotopes have been measured, the internally normalized values can be calculated for all isotopes at once:

```
>>> msrs = (1000000, 250000, 2600, 10000, 2000)
>>> norm_msrs = (msrs[0], msrs[3])  # Ni-58 and Ni-62
>>> ini.iso_int_norm("Ni", ("Ni-58", "Ni-62"), msrs, norm_msrs, delta_factor=1000)
array([ 0.00000000e+00,  5.14586471e+02, -4.49223918e+02,  2.22044605e-13,
        7.41295081e+02])
```

  As expected, the internally normalized values for ^{58}Ni and ^{62}Ni are zero within numerical precisions.

5.2 Configurations

5.2.1 Normalization isotope

Throughout this package, you are able to set isotope ratios by selecting an element for the denominator. In these cases, `iniabu` will by default select the most abundant isotope of this element as the normalizing one. However, you can configure `iniabu` to overwrite this behavior. This especially handy if you want to normalize ratios by default to

another isotope. For example, barium is often normalized to ^{136}Ba instead of the most abundant ^{138}Ba. In order to set this isotope as the main one, you can run the following:

```
>>> from iniabu import ini
>>> ini.norm_isos = {"Ba": "Ba-136"}
```

The `ini.norm_isos` property holds a dictionary with user defined normalization / main isotopes. You can add more isotopes after the fact:

```
>>> ini.norm_isos = {"Si": "Si-29"}
>>> ini.norm_isos
{'Ba': 'Ba-136', 'Si': 'Si-29'}
```

This would now hold both normalization isotopes that were defined. To reset the dictionary, run:

```
>>> ini.reset_norm_isos()
```

5.3 Background information

This section serves to further explain details of the databases and notations in a scientific concept. Here, background information is given that can help the user to better understand the various elements of the package and the logic behind it. Usage of the module is not discussed here.

5.3.1 Databases

Currently, the default database that is loaded is "lodders09".

"lodders09"

The database named "lodders09" is based on the work by Lodders et al. (2009). This database is **loaded by default**, unless a different database is specified.

The measurements loaded when "lodders09" is selected are the data in Table 10 for elements and isotopes. The elemental abundances are simply gained by adding up the isotopic abundances. Note that this introduces a total abundance of Si that is 999700, which is within uncertainties equal to 10^6.

The solar abundances loaded with Lodders et al. (2009) are nuclide abundances 4.56 Ga ago.

Note: The nuclide abundance of ^{138}La in Table 10 of Lodders et al. (2009) is given as 0.000. This seems to be an error originating too few significant figures. Using the atom percentages and the nuclide abundance for ^{139}La, we calculated a nuclide abundance of 0.0004 for ^{138}La and used this calculated abundance for our database.

"asplund09"

The database named "asplund09" is based on the work by Asplund et al. (2009).

The loaded elemental abundances are taken from Table 1 in Asplund et al. (2009) and represent the present-day solar photosphere (column "Photosphere"). The isotope abundances are taken from Table 3 in Asplund et al. (2009) and are the representative isotopic abundance fractions in the Solar System. According to the authors, most isotopic values are taken from Rosman & Taylor (1998) with some updates discussed in Section 3.10 of Asplund et al. (2009).

"nist"

The database named "nist" is based on the online-available abundance table of the National Institute of Standards and Technology. The database can be found here.

To directly quote the database: "In the opinion of the Subcommittee for Isotopic Abundance Measurements (SIAM), these values represent the isotopic composition of the chemicals and/or materials most commonly encountered in the laboratory. They may not, therefore, correspond to the most abundant natural material. The uncertainties listed in parenthesis cover the range of probable variations of the materials as well as experimental errors. These values are consistent with the values published in Isotopic Compositions of the Elements 2009."

More details can be found here.

5.3.2 Notations

δ-values

The δ-value of a given isotope ratio, generally used in cosmo- and geochemistry, is defined as:

$$\delta\left(\frac{^iX}{^jX}\right) = \left(\frac{\left(\frac{^iX}{^jX}\right)_{\text{measured}}}{\left(\frac{^iX}{^jX}\right)_{\text{solar}}} - 1\right) \times f$$

Here, the measured isotope ratio of element X and isotopes i and j represents the ratio as measured in a stardust grain or as modeled in a stellar model. The solar isotope ratio for the same isotope ratio is the one chosen from the database.

Subtracting unity form the ratio of ratios determines the deviation of the measurement from the solar abundance.

Note: The part of the equation in parenthesis should correctly be referred to as the δ-value, i.e., the δ-value is defined when setting $f = 1$.

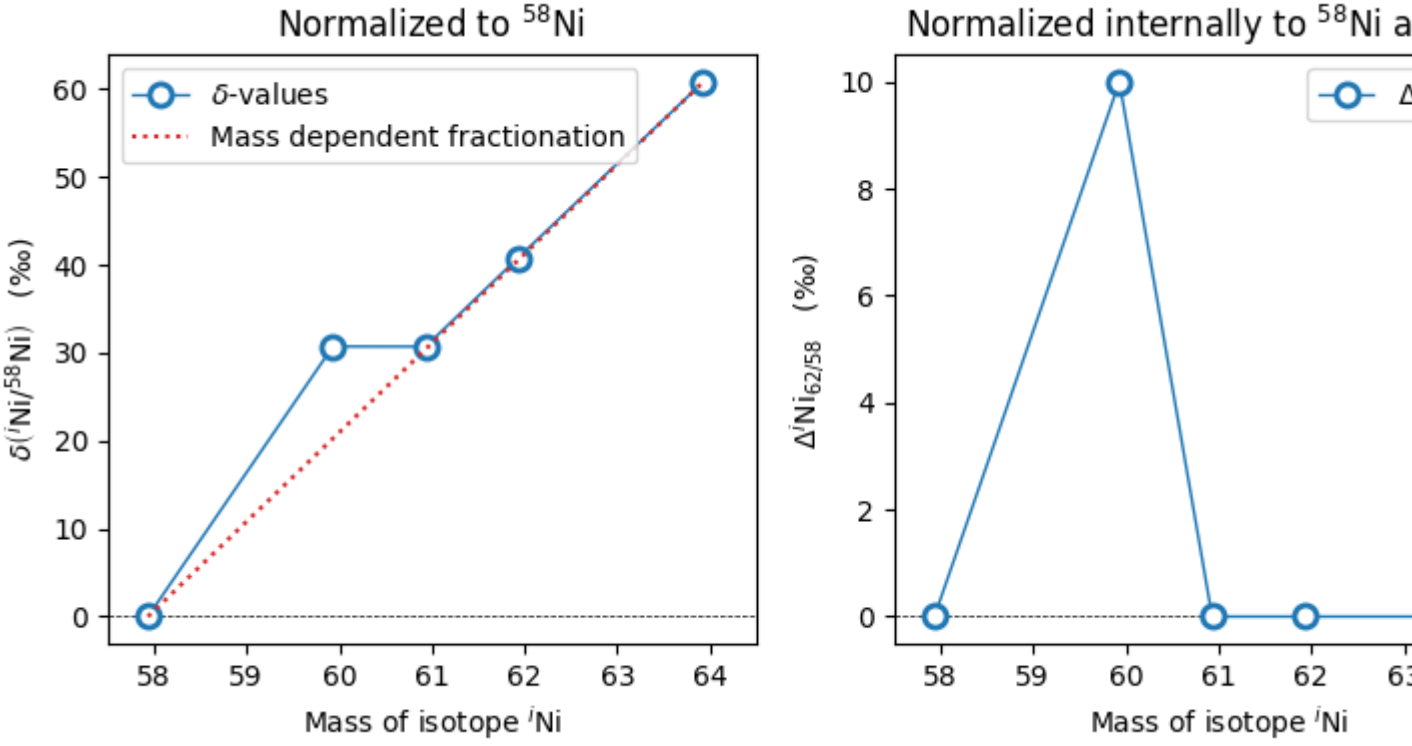
This is important to remember. However, many measurements, especially of stardust, are expressed in parts per thousand or per mil. This means that the δ-value must be multiplied by a factor $f = 1000$.

On the other hand, bulk measurements of meteorites generally detect smaller deviations from solar. Thus, such measurements are often expressed in so-called ϵ- or μ-values. These generally only differ from the δ-value by using a different factor f. The table below gives an overview of different notations and the respective f-values:

Notation	f-value
absolute deviation	1
%, percent	100
‰, per mil	1,000
ϵ, parts per ten thousand	10,000
μ, parts per one-hundred thousand	100,000
ppm, parts per million	1,000,000
ppb, parts per billion	1,000,000,000
ppt, parts per trillion	1,000,000,000,000

Internal normalization

In cosmo- and geochemistry, measured isotope ratios are often internally normalized. This is especially true for measurements that suffer from mass-dependent fractionation.



Above figure shows an example of the two normalization scenarios. On the left side is the regular δ-value notation as described above. As the normalization isotope, ^{58}Ni is chosen. The red, dashed line shows the internal, mass-dependent fractionation that was introduced into the system artificially. Clearly, ^{60}Ni shows some positive deviation from this line. After internal normalization, a clear excess in ^{60}Ni can be seen in the figure.

Internal normalization (right side) normalizes the same dataset to a second isotope. Here, ^{62}Ni is chosen. Assuming that any anomaly in ^{62}Ni is due to mass-dependent fractionation, all isotope ratios can be corrected for this mass-dependent fractionation. To do so, a mass-dependent fractionation law must be applied. These, internally normalized values, if expressed in permil, are often described with a capital delta (Δ).

Note: The same pre-factors as discussed above are applied for internal normalization. Often, measurements obtained using inductively-coupled plasma mass spectrometry (ICP-MS) are internally normalized and results are expressed in ϵ- (parts per 10,000) or μ-values (parts per 100,000). Note that the same notation is frequently used for both normalizations.

A detailed description on mass fractionation laws can be found in Dauphas and Schauble (2016).

In the `iniabu` package, corrections using an exponential (default) and linear law can be applied.

The **exponential law**, which is applied by default, assumes an exponential relation for the mass dependent mass fractionation. Let us assume the example from the above figure. The major normalization isotope ^{j}Ni here is ^{58}Ni, the minor normalization isotope ^{i}Ni is ^{62}Ni. For a given sample, an exponential factor β can be calculated as:

$$\beta = \frac{\log(^i\text{Ni}/^j\text{Ni})_\text{sample} / \log(^i\text{Ni}/^j\text{Ni})_\text{solar}}{\log(m_i/m_j)}$$

Using this exponential factor, the mass-dependent fractionation corrected value of an isotope ratio of interest, e.g.,

^{x}Ni/^{j}Ni can be calculated as:

$$\left(\frac{^{x}\mathrm{Ni}}{^{j}\mathrm{Ni}}\right)^{*}_{\mathrm{sample}} = \frac{(^{x}\mathrm{Ni}/^{j}\mathrm{Ni})_{\mathrm{sample}}}{(m_x/m_j)^{\beta}}$$

Using this corrected ratio, the Δ-value can be calculated as:

$$\Delta^{x}\mathrm{Ni}_{i/j} = \left(\frac{(^{x}\mathrm{Ni}/^{j}\mathrm{Ni})^{*}_{\mathrm{sample}}}{(^{x}\mathrm{Ni}/^{j}\mathrm{Ni})_{\mathrm{solar}}} - 1\right) \times k$$

Here k is the delta factor and defines the unit as described above for δ-values.

The **linear law** to correct for mass-dependent fractionation can be calculated as following:

$$\Delta^{x}\mathrm{Ni}_{i/j} = \delta^{x}\mathrm{Ni}_{j} - \frac{m_j - m_x}{m_j - m_i} \times \delta^{i}\mathrm{Ni}_{j}$$

Here, $^{x}\mathrm{Ni}_{j}$ is short for the ratio $^{x}\mathrm{Ni}/^{j}\mathrm{Ni}$.

The delta factor k is part of the δ-value calculation. With the linear law, values smaller than -(delta factor) are theoretically possible, however, such values are unphysical. The `iso_int_norm` routine automatically detects such values and sets them to the minimal possible value of -(delta factor).

5.4 License

Also, for each author's protection and ours, we want to make certain

that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software

patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and

modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains

a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's

source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion

of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it,

under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

> a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
>
> b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
>
> c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program

except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not

signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the

Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise

of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent

infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in

certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions

of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free

programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY

FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest

possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest

to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

5.5 API Reference

The API reference is split up into four modules. The main `IniAbu` class, with which the user will most likely interact, is described in *Initial Abundance Main Class*.

Querying classes for elements and isotopes are respectively described in *Element Queries* and *Isotope Queries*.

Finally, utility functions and classes are described in *Utility classes and functions*.

Contents:

5.5.1 Initial Abundance Main Class

`IniAbu` Initial Abundance Class

class iniabu.main.**IniAbu**(*database='lodders09'*, *unit='num_lin'*)

Initialize the IniAbu class.

By default, the `lodders09` database is read in. Available databases are:

- `asplund09`: Asplund et al. (2009), doi: 10.1146/annurev.astro.46.060407.145222
- `lodders09`: Lodders et al. (2009), doi: 10.1007/978-3-540-88055-4_34
- `nist`: Current (as of 2020) NIST isotopic abundances.

For detailed examples, see: https://iniabu.readthedocs.io/

Example:

```
>>> from iniabu import ini
>>> ini.iso["Ne"].name
['Ne-20', 'Ne-21', 'Ne-22']
>>> ini.iso["Ne"].abu_solar
array([3060000.,    7330.,  225000.])
>>> ini.iso_delta("Si-30", "Si-28", 0.02)
-403.23624595469255
```

database

Get / Set the current database.

Setting a new database does not change the units that are currently loaded. You will get a message printed on what these units are.

Setter Database to set.

Type str

Returns Name of the loaded database.

Return type str

Example:

```
>>> from iniabu import ini  # loads with default ("lodders09")
>>> ini.database = "nist"  # change database to "nist"
>>> ini.database
'nist'
>>> ini.database = "lodders09"  # simple switch back!
```

ele

Get information for a specific element.

Calls the :class‘iniabu.elements.Elements‘. This handler represents a convenient way to dig through elemental information. More information and a full list of properties can be found in the *Elements* class.

Returns Returns a ProxyList initialized with the required element

Return type class

Example:

```
>>> from iniabu import ini
>>> # get the solar abundance of silicon
>>> ini.ele["Si"].abu_solar
999700.0
```

```
>>> # get a numpy array of the solar abundance of two elements
>>> ini.ele[["Fe", "Ni"]].abu_solar
array([847990.,  49093.])
```

```
>>> # get a list of all atomic numbers for isotopes of helium
>>> ini.ele["He"].iso_a
array([3, 4])
```

```
>>> # similarly, query isotopes relative abundances, and solar abundances
>>> ini.ele["He"].iso_abu_rel
array([1.66000e-04, 9.99834e-01])
>>> ini.ele["He"].iso_abu_solar
array([1.03e+06, 2.51e+09])
```

ele_bracket(*nominator*, *denominator*, *value*, *mass_fraction=None*)

Calculate the bracket ratio for a given element ratio and a value.

The bracket notation is the usual astronomy / logarithmic ratio, defined as:

result = log10(measured value) - log10(solar ratio)

Nominator and denominator have the same restrictions as for the `ele_ratio` method. If one element ratio is defined but multiple values, the isotope ratio for all values is calculated and returned as a numpy array. If more than one element ratio is defined, the same number of values must be supplied as there are element ratios defined.

Parameters

- **nominator** (*str, list*) – Element(s) in nominator.
- **denominator** (*str, list*) – Element(s) in denominator.
- **value** (*float, ndarray*) – Value(s) to calculate bracket notation value with respect to.
- **mass_fraction** (*bool*) – Are the given values in mass fractions? Defaults to None, which makes it dependent on the units that are currently loaded. The loaded setting can be overwritten by setting *mass_fraction=True* or *mass_fraction=False*.

Returns Bracket notation expression of given values with respect to the solar system abundances.

Return type float,ndarray

Raises **ValueError** – Number of element ratios and number of values supplied are mismatched.

Example:

```
>>> from iniabu import ini
>>> ini.ele_bracket("Ne", "Si", 33)
1.0008802726402624
```

ele_delta (*nominator*, *denominator*, *value*, *mass_fraction=None*, *delta_factor=1000.0*)
Calculate the delta-value for a given element ratio and a value.

The delta-value is defined as:

result = (measured value / solar ratio - 1)

By default, the delta-value is multiplied by 1000, thus, expressing it in permil. Other factors can be chosen.

Nominator and denominator have the same restrictions as for the `ele_ratio` method. If one element ratio is defined but multiple values, the isotope ratio for all values is calculated and returned as a numpy array. If more than one element ratio is defined, the same number of values must be supplied as there are element ratios defined.

Parameters

- **nominator** (*str, list*) – Element(s) in nominator.
- **denominator** (*str, list*) – Element(s) in denominator.
- **value** (*float, ndarray*) – Value(s) to calculate delta-value with respect to.
- **mass_fraction** (*bool*) – Are the given values in mass fractions? Defaults to None, which makes it dependent on the units that are currently loaded. The loaded setting can be overwritten by setting *mass_fraction=True* or *mass_fraction=False*.
- **delta_factor** – What value should the delta value be multiplied with? Defaults to 1000 to return results in permil.
- **delta_factor** – float

Returns Delta-values of given values with respect to the solar system abundances, multiplied by delta_factor (by default, returns delta-values in permil).

Return type float,ndarray

Raises **ValueError** – Number of element ratios and number of values supplied are mismatched.

Example:

```
>>> from iniabu import ini
>>> ini.ele_delta("Ne", "Si", 3.4)
32.39347210030586
```

ele_dict
Get the element dictionary.

The dictionary keys are element symbols, e.g., “H”. The entries for the element dictionary are a list containing the following entries (in order):

- Solar abundance

- ndarray with mass numbers of all isotopes
- ndarray with relative abundances of all isotopes
- ndarray with solar abundances of all isotopes

Returns Element dictionary

Return type dict

ele_dict_log
Get the element dictionary with logarithmic solar abundances.

The dictionary keys are element symbols, e.g., "H". The entries for the element dictionary are a list containing the following entries (in order):

- Solar abundance (log)
- ndarray with mass numbers of all isotopes
- ndarray with relative abundances of all isotopes
- ndarray with solar abundances of all isotopes (log)

Returns Element dictionary

Return type dict

ele_dict_mf
Get the element dictionary with mass fractions.

The dictionary keys are element symbols, e.g., "H". The entries for the element dictionary are a list containing the following entries (in order):

- Solar abundance (mass fractions)
- ndarray with mass numbers of all isotopes
- ndarray with relative abundances of all isotopes
- ndarray with solar abundances of all isotopes (mass fractions)

Returns Element dictionary

Return type dict

ele_ratio(*nominator*, *denominator*, *mass_fraction=None*)
Get the ratios of given elements.

Nominator and denominator can be element names or lists of element names (if more than one ratio should be calculated). If the denominator is a list, its length must be identical to the list in the nominator.

Parameters

- **nominator** (*str, list*) – Element or list of elements in nominator of ratio.
- **denominator** (*str, list*) – Element or list of elements in denominator of ratio.
- **mass_fraction** (*bool*) – Are the given values in mass fractions? Defaults to None, which makes it dependent on the units that are currently loaded. The loaded setting can be overwritten by setting *mass_fraction=True* or *mass_fraction=False*.

Returns The element ratio or a numpy array of the requested ratios.

Return type float,ndarray

Raises **ValueError** – Denominator is a list with more than one entry and does not have the same length as the nominator.

Example:

```
>>> from iniabu import ini
>>> # Calculate H/He ratio
>>> ini.ele_ratio("H", "He")
10.314692775474606
```

```
>>> # Calculate same ratio as mass fraction
>>> ini.ele_ratio("H", "He", mass_fraction=True)
2.597460199709773
```

```
>>> # Calculate ratios with multiple elements
>>> ini.ele_ratio(["H", "He", "Al"], ["Si"])
array([2.59082755e+04, 2.51178354e+03, 8.46253876e-02])
```

```
>>> # Multiple ratios at the same time
>>> ini.ele_ratio(["H", "He"], ["He", "H"])
array([10.31469278,  0.09694908])
```

```
>>> # The result when the solar abundance of an element is not avaialble
>>> ini.database = "nist"
>>> ini.ele_ratio("H", "He")
nan
```

```
>>> ini.database = "lodders09"  # set back to default and check again
>>> ini.ele_ratio("H", "He")
10.314692775474606
```

iso

Get information for a specific isotope.

Calls the :class‘iniabu.isotopes.Isotopes‘. This handler represents a convenient way to dig through isotopic information. More information and a full list of properties can be found in the *Isotopes* class.

Returns Returns a ProxyList initialized with the required element

Return type class

Example:

```
>>> from iniabu import ini
>>> # get the solar abundance of Si-28
>>> ini.iso["Si-28"].abu_solar
922000.0
```

```
>>> # get a numpy array of the solar abundance of two isotopes
>>> ini.iso[["Fe-56", "Ni-60"]].abu_solar
array([778000.,  12900.])
```

```
>>> # similarly, query relative abundance(s) of isotope(s)
>>> ini.iso["He-4"].abu_rel
0.999834
```

(continues on next page)

(continued from previous page)

```
>>> ini.iso[["H-2", "He-3"]].abu_rel
array([1.94e-05, 1.66e-04])
```

iso_bracket (*nominator*, *denominator*, *value*, *mass_fraction=None*)
Calculate the bracket ratio for a given isotope ratio and a value.

The bracket notation is the usual astronomy / logarithmic ratio, defined as:

result = log10(measured value) - log10(solar ratio)

Nominator and denominator have the same restrictions as for the `ele_ratio` method. If one isotope ratio is defined but multiple values, the isotope ratio for all values is calculated and returned as a numpy array. If more than one isotope ratio is defined, the same number of values must be supplied as there are isotope ratios defined.

Parameters

- **nominator** (*str, list*) – Isotope(s) in nominator.
- **denominator** (*str, list*) – Isotope(s) in denominator.
- **value** (*float, ndarray*) – Value(s) to calculate bracket notation value with respect to.
- **mass_fraction** (*bool*) – Are the given values in mass fractions? Defaults to None, which makes it dependent on the units that are currently loaded. The loaded setting can be overwritten by setting *mass_fraction=True* or *mass_fraction=False*.

Returns Bracket notation expression of given values with respect to the solar system abundances.

Return type float,ndarray

Raises **ValueError** – Number of element ratios and number of values supplied are mismatched.

Example:

```
>>> from iniabu import ini
>>> ini.iso_bracket("Ne-21", "Ne-20", 2.397)
3.0002854858741057
```

iso_delta (*nominator*, *denominator*, *value*, *mass_fraction=None*, *delta_factor=1000.0*)
Calculate the delta-value for a given isotope ratio and a value.

The delta-value is defined as:

result = (measured value / solar ratio - 1)

By default, the delta-value is multiplied by 1000, thus, expressing it in permil. Other factors can be chosen.

Nominator and denominator have the same restrictions as for the `ele_ratio` method. If one isotope ratio is defined but multiple values, the isotope ratio for all values is calculated and returned as a numpy array. If more than one isotope ratio is defined, the same number of values must be supplied as there are isotope ratios defined.

Parameters

- **nominator** (*str, list*) – Isotope(s) in nominator.
- **denominator** (*str, list*) – Isotope(s) in denominator.

- **value** (*float,ndarray*) – Value(s) to calculate delta-value with respect to.
- **mass_fraction** (*bool*) – Are the given values in mass fractions? Defaults to None, which makes it dependent on the units that are currently loaded. The loaded setting can be overwritten by setting *mass_fraction=True* or *mass_fraction=False*.
- **delta_factor** – What value should the delta value be multiplied with? Defaults to 1000 to return results in permil.
- **delta_factor** – float

Returns Delta-values of given values with respect to the solar system abundances, multiplied by delta_factor (by default, returns delta-values in permil).

Return type float,ndarray

Raises **ValueError** – Number of isotope ratios and number of values supplied are mismatched.

Example:

```
>>> from iniabu import ini
>>> ini.iso_delta("Ne-22", "Ne-20", 0.07, delta_factor=10000)
-479.9999999999993
```

```
>>> # For more than 1 ratio
>>> nominator_isos = ["Ne-21", "Ne-22"]
>>> values = [0.01, 0.07]  # values to compare with
>>> ini.iso_delta(nominator_isos, "Ne-20", values, delta_factor=10000)
array([31746.24829468,  -480.        ])
```

iso_dict

Get the isotope dictionary.

The dictionary keys are isotope symbols, e.g., "H-1". The entries for the isotope dictionary are a list containing the following entries (in order):

- Relative abundance
- Solar abundance

Returns Isotope dictionary

Return type dict

iso_dict_log

Get the isotope dictionary with logarithmic solar abundances.

The dictionary keys are isotope symbols, e.g., "H-1". The entries for the isotope dictionary are a list containing the following entries (in order):

- Relative abundance
- Solar abundance (log)

Returns Isotope dictionary

Return type dict

iso_dict_mf
Get the isotope dictionary in mass fractions.

The dictionary keys are isotope symbols, e.g., "H-1". The entries for the isotope dictionary are a list containing the following entries (in order):

- Relative abundance
- Solar abundance (mass fractions)

Returns Isotope dictionary

Return type dict

iso_int_norm(*nominator*, *norm_isos*, *sample_values*, *sample_norm_values*, *delta_factor=10000*, *law='exp'*)
Calculate internally normalized value for isotope data with respect to solar.

The internally normalized value requires two normalizing isotopes. This normalization ratios the value to one normalization isotope and uses the second one to correct for mass-dependent fractionation. Details can be found in the background section of the documentation.

Note: A *mass_fraction* toggle, as for other routines, is useless here. Internal normalization, by definition, removes any fractionation effects due to mass.

An elemental routine analogous to this one does not make sense since elemental ratios do usually show other effects than mass dependent ones.

Parameters

- **nominator** (*str or tuple/list(str)*) – Name of the nominator isotope(s) to be used. Multiple can be selected at once by giving them as a list.
- **norm_isos** (*tuple/list(str, str)*) – The names of the normalizing isotopes. First is the major normalizing isotope, i.e., the one that the sample value is ratioed to. The second one is the isotope that is used for correcting mass fractionation effects.
- **sample_values** (*float, ndarray/tuple/list(floats)*) – The sample's value(s) for the nominator isotope. Shape must match the *nominator* name and values must be in the same order.
- **sample_norm_values** (*tuple/list/ndarray(float, float)*) – The sample's values for the normalization isotopes. Same order as the normalization isotope names.
- **delta_factor** (*float*) – What factor should the normalization be multiplied? Defaults to 10000 (for internally normalized epsilon values).
- **law** (*str*) – Normalization law to use: Either "exp" for exponential low or "lin" for linear law. Defaults to "exp"

Returns Internally normalized delta-values multiplied with given factor. Returns as many values as given in nominator / sample_values.

Return type float, ndarray(float)

Raises **ValueError** – Input values are mismatched, selected law is not valid.

Example:

```
>>> from iniabu import ini
>>> # define input values
>>> norm_isos = ("Ni-58", "Ni-60") # normalizing isotopes
>>> sample_vals = (1., 0.4, 0.15, 0.5, 0.3)  # measured isotope abundances
>>> sample_norm_vals = (sample_vals[0], sample_vals[1])  # Ni-58, Ni-60
>>> # get the internally normalized values for all nickel isotopes
>>> ini.iso_int_norm("Ni", norm_isos, sample_vals, sample_norm_vals)
array([     0.        ,      0.        ,  75068.93030287,  77568.12815864,
       189333.87185102])
```

iso_ratio(*nominator*, *denominator*, *mass_fraction=None*)

Get the ratios of given isotopes.

Grabs the isotope ratios for nominator / denominator. If a list of nominator isotopes is given but only one denominator isotope, the ratio with that denominator is formed for each isotope. If both parameters are given as lists, they must be of equal length.

Parameters

- **nominator** (*str, list*) – Isotope / List of isotopes for nominator, in form: "Si-29". If an element is given, all isotopes of this element are used. Lists of elements are not allowed.
- **denominator** (*str, list*) – Isotope / List of isotopes for denominator, in form: "Si-29". Alternatively, an element can be given, i.e., "Si". In that case, the most abundant isotope is chosen. Lists of elements are not allowed.
- **mass_fraction** (*bool*) – Are the given values in mass fractions? Defaults to None, which makes it dependent on the units that are currently loaded. The loaded setting can be overwritten by setting *mass_fraction=True* or *mass_fraction=False*.

Returns The isotope ratio or a numpy array of the requested ratios.

Return type float,ndarray

Raises **ValueError** – Denominator is a list with more than one entry and does not have the same length as the nominator.

Example:

```
>>> from iniabu import ini
>>> # calculate Ne-21 / Ne-20 isotope ratio
>>> ini.iso_ratio("Ne-21", "Ne-20")
0.002395424836601307
```

```
>>> # calculate isotope ratios for all Ne isotopes versus Ne-20
>>> ini.iso_ratio("Ne", "Ne-20")
array([1.        , 0.00239542, 0.07352941])
```

```
>>> # Isotope ratios for Ne-21 and Ne-22 versus most abundant Ne isotope
>>> ini.iso_ratio(["Ne-21", "Ne-22"], "Ne")
array([0.00239542, 0.07352941])
```

```
>>> # repeat this calculation assuming mass fractions
>>> ini.iso_ratio(["Ne-21", "Ne-22"], "Ne", mass_fraction=True)
array([0.00251541, 0.08088125])
```

```
>>> from iniabu import inimf
>>> # calculate Ne-21 / Ne-20 isotope ratio using mass fractions
>>> inimf.iso_ratio("Ne-21", "Ne-20")
0.002515409891030499
```

```
>>> # calculate the same ratio in number fractions
>>> inimf.iso_ratio("Ne-21", "Ne-20", mass_fraction=False)
0.002395424836601307
```

norm_isos

Get / Set user defined normalization isotopes.

Note: If you set the *norm_isos* multiple times, it will not be overwritten. Keys will rather be added to the dictionary. To reset it, please use the function *ini.reset_norm_isos()*.

Setter A dictionary with your normalization isotope per element.

Returns The dictionary with user defined normalization isotopes.

Example:

```
>>> from iniabu import ini
>>> ini.norm_isos
{}
>>> ini.norm_isos = {"Ba": "Ba-136"}
>>> ini.norm_isos
{'Ba': 'Ba-136'}
>>> ini.norm_isos = {"Si": "Si-29"}
>>> ini.norm_isos
{'Ba': 'Ba-136', 'Si': 'Si-29'}
>>> ini.reset_norm_isos()
>>> ini.norm_isos
{}
```

reset_norm_isos()

Reset the user defined normalization isotopes.

This will result in the normalization isotopes to be simple a empty dictionary again.

Example:

```
>>> from iniabu import ini
>>> ini.norm_isos = {"Ba": "Ba-136"}
>>> ini.norm_isos
{'Ba': 'Ba-136'}
>>> ini.reset_norm_isos()
>>> ini.norm_isos
{}
```

unit

Get / Set the unit for the solar abundances.

Routine to easily switch the database between the **default** linear number abundances, normed to Si with an abundance of 1e6 (`num_lin`, typically used in cosmo- and geochemistry studies), the logarithmic

(num_log, typically used in astronomy) abundance units, normed to H as 12, or mass fractions massf, normed such that all elements sum up to unity.

Setter Unit to set, either "num_lin" (default), "num_log", or "mass_fraction".

Type str

Returns Currently set unit.

Return type str

Example:

```
>>> from iniabu import ini  # loads with default linear units
>>> ini.unit = "num_log"  # set logarithmic abundance unit
>>> ini.unit
'num_log'
>>> ini.ele["H"].abu_solar
12.0
```

```
>>> ini.unit = "num_lin"  # set back to default
>>> ini.unit
'num_lin'
```

5.5.2 Element Queries

Elements Querying Class

class iniabu.elements.**Elements**(*parent*, *eles*, *unit='num_lin'*, **args*, ***kwargs*)

Class representing the elements.

This is mainly a list to easily interact with the *parent._ele_dict* dictionary.

Example:

```
>>> from iniabu import ini
>>> ini.unit
'num_lin'
>>> element = ini.ele["Si"]
>>> element.abu_solar
999700.0
```

Warning: This class should NOT be manually created by the user. It is designed to be initialized by iniabu.IniAbu.

abu_solar

Get solar abundance of element(s).

Returns the solar abundance of the selected element(s). Returns the result either as a float or as a numpy ndarray. Note: Not all databases contain this information. If the information is not available, these values will be filled with np.nan.

Returns Solar abundance of element(s)

Return type float,ndarray

iso_a
Get the atomic number(s) of all isotopes.

Returns the atomic number(s) of all isotopes of this element as a numpy integer ndarray. If more than one element is selected, a list of numpy integer arrays is returned.

Returns Atomic numbers of all isotopes

Return type ndarray,list<ndarray>

iso_abu_rel
Get relative abundance of all isotopes.

Returns a list with the relative abundances of all isotopes of the given element. If more than one element is selected, a list of numpy float ndarrays is returned. Note: All relative abundances sum up up to unity. If you are using "mass_fractions" as units, relative abundances will also be in mass fractions.

Returns Relative abundance of all isotopes

Return type ndarray,list<ndarray>

iso_abu_solar
Get solar abundances of all isotopes.

Returns a list with the solar abundances of all isotopes of the given element. If more than one element is selected, a list of numpy float ndarrays is returned. Note: Not all databases contain this information. If the information is not available, these values will be filled with `np.nan`.

Returns Relative abundance of all isotopes

Return type ndarray,list<ndarray>

mass
Get the mass of an element.

Returns the mass of an element depending on the specified composition. The mass is calculated as the weighted sum of the individual isotope masses, weighted by there respective abundances.

Returns Mass of an element.

Return type float,ndarray<float>

name
Get the name of an element.

Returns Name of the set element(s).

Return type str, list(str)

z
Get the number of protons for the element.

Returns Number of protons for the set element(s).

Return type int, ndarray<int>

5.5.3 Isotope Queries

`Isotopes` Querying Class

class `iniabu.isotopes.`**Isotopes**(*parent*, *isos*, *unit='num_lin'*, **args*, ***kwargs*)
Class representing the isotopes.

This is mainly a list to easily interact with the *parent._iso_dict* dictionary.

Example:

```
>>> from iniabu import ini
>>> isotope = ini.iso["Si-28"]
>>> isotope.abu_rel
0.9223
```

Note: You can also call isotopes using alternative spellings, e.g., "28Si" or "Si28".

Warning: This class should NOT be manually created by the user. It is designed to be initialized by `iniabu.IniAbu`

a
Get total number of nucleons for given isotope.

Returns the total number of nucleons for the given isotope. Sure, this is already passed as an argument in the isotope name, however, might be useful for plotting to have a return for it.

Returns Mass number of isotope

Return type int, ndarray<int>

a_all
Get total number of nucleons for all available isotope(s).

Returns the total number of nucleons for the given isotope. Sure, this is already passed as an argument in the isotope name, however, might be useful for plotting to have a return for it. All available isotopes means stable and unstable.

Returns Mass number of isotope

Return type int, ndarray<int>

abu_rel
Get relative abundance of isotope(s).

Returns the relative abundance of the selected isotope(s). Returns the result either as a `float` or as a numpy `ndarray`. Note: All relative abundances sum up up to unity. If you are using "mass_fractions" as units, relative abundances will also be in mass fractions.

Returns Relative abundance of isotope(s)

Return type float,ndarray

abu_rel_all
Get relative abundance of all available isotope(s).

Returns the relative abundance of the selected isotope(s). Returns the result either as a `float` or as a numpy `ndarray`. Note: All relative abundances sum up up to unity. If you are using "mass_fractions" as units, relative abundances will also be in mass fractions. All available isotopes means stable and unstable.

Returns Relative abundance of isotope(s)

Return type float,ndarray

abu_solar
Get solar abundance of isotope(s).

Returns the solar abundance of the selected isotope(s). Returns the result either as a `float` or as a numpy `ndarray`. Note: Not all databases contain this information. If the information is not available, these values will be filled with `np.nan`.

Returns Solar abundance of isotope(s)

Return type float,ndarray

abu_solar_all
Get solar abundance of all available isotope(s).

Returns the solar abundance of the selected isotope(s). Returns the result either as a `float` or as a numpy `ndarray`. Note: Not all databases contain this information. If the information is not available, these values will be filled with `np.nan`. All available isotopes means stable and unstable.

Returns Solar abundance of isotope(s)

Return type float,ndarray

mass
Get the mass of an isotope.

Returns Mass of an isotope.

Return type float,ndarray<float>

mass_all
Get the mass of all available isotopes.

All available isotopes means stable and unstable.

Returns Mass of an isotope.

Return type float,ndarray<float>

name
Get the name of an isotope.

If an alternative spelling was used to call the isotope, e.g., "Si28" or "28Si", the name will still be returned as "Si-28", which is the default for *iniabu*.

Returns Name of the set isotope(s).

Return type str, list(str)

name_all
Get the names of all available isotope.

If an alternative spelling was used to call the isotope, e.g., "Si28" or "28Si", the name will still be returned as "Si-28", which is the default for *iniabu*. All available isotopes means stable and unstable.

Returns Name of the set isotope(s).

Return type str, list(str)

z
Get the number of protons for the isotopes.

Returns Number of protons for the set isotope(s).

Return type int, ndarray<int>

z_all
Get the number of protons for the isotopes.

All available isotopes means stable and unstable.

Returns Number of protons for the set isotope(s).

Return type int, ndarray<int>

5.5.4 Utility classes and functions

ProxyList

class iniabu.utilities.**ProxyList**(*parent*, *proxy_cls*, *valid_set*, **args*, ***kwargs*)

Proxy for accessing elements and isotopes as lists.

This class is inspired by a class with the same name from the project `InstrumentKit` by Galvant Industries. It is used to generate lists of objects. The valid keys are defined by the *valid_set* initialization parameter. This allows generating a single property for elements and isotopes to access them.

InstrumentKit

get_all_available_isos()

iniabu.utilities.**get_all_available_isos**(*ele*)

Get all available isotopes of a given element, stable and unstable.

This is particularly interesting if we want to know the mass of unstable isotopes.

Parameters **ele** (*str*) – Element name

Returns All isotopes of the element as a list.

Return type list(str)

get_all_stable_isos()

iniabu.utilities.**get_all_stable_isos**(*ini*, *ele*)

Get all isotopes of a given element.

Parameters

- **ini** (IniAbu) – Initialized iniabu instance
- **ele** (*str*) – Element name

Returns All isotopes of the element as a list.

Return type list(str)

item_formatter()

iniabu.utilities.**item_formatter**(*iso: str*) → str

Transform *iso* into correct format, e.g,. from *46Ti* to *Ti-46*.

Also appropriately capitalizes isotopes and elements.

Supported formats: - *46Ti* - *Ti46* - *Ti-46*

Parameters **iso** – Isotope as string or element name.

Returns iso, but in transformed notation and capitalized

linear_units()

iniabu.utilities.**linear_units**(*ini*, *mass_fraction*)

Context manager to turn current instants units linear if logarithmic.

This is used mainly for ratio calculation, since logarithmic cannot be ratioed to each other.

Parameters

- **ini** (*IniAbu*) – Initialized iniabu instance
- **mass_fraction** (*bool or None*) – Mass fraction variable passed on from last routine

Yield *ini* as with adjusted units (if necessary)

Ytype *IniAbu* instance

make_iso_dict()

iniabu.utilities.**make_iso_dict**(*element_dict*)

Make an isotope dictionary from an element dictionary.

Parameters **element_dict** (*dict*) – Element dictionary.

Returns Isotope dictionaries with same abundances as element dictionary.

Return type dict

make_log_abu_dict()

iniabu.utilities.**make_log_abu_dict**(*element_dict*)

Make element and isotope dictionaries for logarithmic abundances.

This routine takes an element dictionary with linear abundances, normed to Si equals 1e6, and returns new dictionaries with logarithmic abundances, normed to log10(Nx/NH) + 12, where Nx is the number abundance of the element in question and NH is the number abundance of hydrogen.

Parameters **element_dict** (*dict*) – Element dictionary.

Returns Element and isotope dictionaries with logarithmic solar abundances.

Return type dict,dict

make_mf_dict()

iniabu.utilities.**make_mf_dict**(*element_dict*)

Make element and isotope dictionaries for mass fractions.

This routine takes an element dictionary with linear abundances, normed to Si equals 1e6, and returns new dictionaries with mass fractions. The mass fraction Xi of an isotope i is defined as Xi = Ni mi / sum(Ni mi). Here, Ni is the number abundance of a element / isotope i and mi is its mass.

Parameters **element_dict** (*dict*) – Element dictionary.

Returns Element and isotope dictionaries with mass fractions.

Return type dict,dict

return_as_ndarray()

iniabu.utilities.**return_as_ndarray**(*val*)
Return the input as a ndarray.

Parameters **val** (*int, float, list, tuple, ndarray*) – Input value.

Returns Array of input value if not an array, otherwise return itself.

Return type ndarray

return_string_as_list()

iniabu.utilities.**return_string_as_list**(*s*)
Return the input as a list.

Parameters **s** (*str, list*) – Input value.

Returns List of input value if not a list, otherwise return itself.

Return type list

return_list_simplifier()

iniabu.utilities.**return_list_simplifier**(*return_list*)
Simplify standard return values.

Specifically written for classes with multiple return types, such as *iniabu.elements.Elements* and *iniabu.isotopes.Isotopes*. If only one entry is in the list, it should not be returned as a list but as a value. Otherwise, return the list.

Parameters **return_list** (*list, ndarray*) – List or numpy array with the value to be returned.

Returns If only one entry in list, return that entry. Otherwise return list.

5.6 Developer Reference

First off, please make sure you have read and understood our code of conduct. We expect everybody to adhere to it. You can find this project's code of condact here.

To get started with developing, fork the github repository and clone it into a local directory. If this is your first time contributing to an open-source project, have a look at these general guidelines.

This project uses fairly tight restrictions in terms of testing and linting. Don't be discouraged if you run into issues. Always feel free to ask questions by raising an issue. Many style guides and ideas here are taken from the Hypermodern Python blog created by Claudio Jolowicz. These blog post, while intense, are an excellent read and are highly recommended.

5.6.1 Dependencies

For full testing of the project, you should have the supported python versions installed. Furthermore, you need to install `nox`, which can be done from the console by typing:

```
$ pip install nox
```

If you completely test your setup with `nox`, dependency installation is not required. If you like to test directly with `pytest`, write your own temporary routines, create examples, etc., you can install the dependencies from your console by typing:

```
$ pip install -r requirements.txt
$ pip install -r dev-requirements.txt
```

5.6.2 Contribution requirements

All code submissions should be tested. The CI requires that all unit tests and lint tests complete successfully before merging into the main branch is allowed.

Coverage:

All code should be tested. Code testing coverage of 100% is required to ensure future integrity.

Docstrings:

The documentation automatically generates the API reference from the supplied docstrings. Please use Sphinx style docstrings to document your routines.

Linting:

All code must adhere to `flake8` specifications, see also *Linting*. This allows for better readability. Even though you won't remember all linting rules, you should go back and fix linting issues after testing. The tests will give you feedback on what to do.

5.6.3 Test driven development

Testing of the `iniabu` package is done using `pytest` and automated using `nox`. To run a full test using `nox`, Python 3.6, 3.7, 3.8, and 3.9, must be available in the environment. A full nox test, which includes linting, safety, and tests can be run from the terminal by typing:

```
$ nox
```

To check wha sessions are implemented, run the following code from your terminal:

```
$ nox -l
```

This will also display information for all sessions implemented in `nox`. You can also check out the `noxfile.py` directly.

Please also check the nox documentation for further options, etc.

The test suite lives in the `tests` folder. This folder mirrors the package structure. In addition, a file named `conftest.py` is used to set fixtures for pytest. This allows for proper initialization of the package with every test.

The iniabu project requires that the whole code base is covered with tests, i.e., that a code coverage of 100% is maintained. Of course, the tests should also be meaningful! This code coverage ensures that future developments do not break other functionalities. More about this can be found in *Testing*.

Example: Bugfix

If a bug is found in the code and reported, a bug fix should be implemented in the following way:

1. Write a test with the wanted outcome and make sure the test suite fails due to the reported bug.
2. Fix the bug in the source code.

3. The bug is fixed once tew new test passes successfully and no other tests were broken.

Formatting with `black`

The iniabu project adopts the default style that is provided by the black python formatter. Their GitHub site describes in detail how to use the formatter. There is really not much to configure.

Formatting with black is implemented via a pre-commit hooks, see section *Pre-commit hooks* for more information. The hook file also specifies the used version of black.

Linting

Linting heavily improves code readability. Please follow all linting guidelines. We use `flake8`. Furthermore, the following additional plugins are used:

- `flake8-bandit` to identify security issues.
- `flake8-black` to check that the codebase is formatted using black.
- `flake8-bugbear` to find additional bugs and design problems.
- `flake8-docstrings` to ensure docsting completeness and consistency.
- `flake8-import-order` to ensure consistent package importing.

Exact linting options are configured in the `.flake8` file. This file also contains comments to better understand the options.

Invoking only linting with nox can be done from the terminal by typing:

```
$ nox -rs lint
```

To fix linting issues, read the output of the linter carefully. If absolutely required, use the `# noqa: err` comment after the line in question to exclude specific linting errors. Replace the `err` part with the error number that was returned by the linter. This should only be used where it makes sense.

Testing

Project testing is done with `pytest`. The following `pytest` plugins are defined in the `dev-requirements.txt` file:

- `pytest-cov` to test code coverage.
- `pytest-mock` to mock out certain parts of the code base.
- `pytest-sugar` to display nicely formatted output.

The `pytest.ini` file configures the testing environment properly. To run tests from the terminal, assuming that all dependencies are installed, type:

```
$ pytest
```

To test the test suite only with `nox`, you can type the following into the terminal:

```
$ nox -rs tests
```

Again, adding the option `-p 3.9` would limit the test to Python 3.9 only.

Hypothesis

Where adequate, make use of the hypothesis package for writing your tests. Have a look at the existing tests for input on what to test for. Hypothesis allows for simple edge case testing and often catches errors that might otherwise go through.

Docstring example testing

As discussed before, docstrings should be used to document every new routine. The docstrings should also contain examples. Check out the source code for examples on how to write them.

Examples should of course represent the behavior of the code. It thus must be written in Python prompt form. For example, look at the following example:

```
>>> from iniabu import ini  # loads with default ("lodders09")
>>> ini.database = "nist"  # change database to "nist"
>>> ini.database
'nist'
```

To ensure that all examples are correct, they can be tested using xdoctest. This is implemented as a `nox` session and can be called by typing the following into your terminal:

```
$ nox -rs xdoctest
```

Note: This is not part of the unit tests and must be called separately. A GitHub action is implemented to specifically run `doctests`.

Safety

Safety is used to check all required dependencies for known security vulnerabilities. To run only `safety` form `nox`, type the following into your terminal:

```
$ nox -rs safety
```

Documentation

The documentation uses `sphinx`. It is automatically built and hosted by readthedocs.io. To locally build the documentation, run the following from your terminal:

```
$ nox -rs docs
```

This will dump the `html` files for the documentation into the `docs/_build` folder. You can now locally browse them.

Pre-commit hooks

Using pre-commit hooks your project can be tested for simple formatting mishaps. These will also be automatically corrected. Here, we use the pre-commit framework. If you want to set up pre-commit hooks, go to the folder and run the following command (after installing pre-commit using pip or pipx):

```
$ pre-commit install
```

This will install the hooks that are defined in *.pre-commit-config.yaml* into your git repository. Note that a fairly standard pre-commit configuration is used. Black is pinned to a specific version, i.e., the same version as in the nox file itself.

5.6.4 Structure of the data tables

All data lives in the `data` subfolder underneath the main package. Aside from the `nist.py` file, all databases contain 2 dictionaries, one for elements and one for isotopes.

Missing values must be denotes as `np.nan`.

`ele_dict` Element dictionary

The element dictionary `ele_dict` is shaped in the following structure:

```
ele_dict = {
             'Symb':
                 [
                     sol_abu_ele,
                     [a1, ..., an],
                     [rel_abu1, ..., rel_abun],
                     [sol_abu1, ..., sol_abun]
                 ],
             ...
             }
```

Here, `Symb` is the element symbol, e.g., `H` for hydrogen. This is the dictionary key. The entry is followed by a list. The entry `sol_abu_ele` is the solar abundance of the element in number fractions normalized such that the solar abundance of Si is 1e6. `a1` to `an` are the atomic mass numbers of the isotopes of this element. `rel_abu1` to `rel_abun` and `sol_abu1` to `sol_abun` are these isotopes relative abundances and solar abundance, respectively. Note that the relative abundances must be normed such that their sum is unity.

`iso_dict` Isotope dictionary

The isotope dictionary `iso_dict` is shaped in the following structure:

```
iso_dict = {
             'Symb-A':
                 [
                     rel_abu,
                     sol_abu
                 ],
             ...
            }
```

Here, *Symb-A* is the key of the dictionary and is composed of the element symbol `Symb` and the isotope's atomic number `A`. A dash separates the two entries. The dictionary entries are `rel_abu` and `sol_abu`, which are the isotopes relative and solar abundance, respectively. The same normalization rules apply as discussed above.

5.6.5 Adding a database

Parser files for individual databases that have already been added were put into the `dev` folder in the repository. Every database added has their datafile in some format and a parser living there. The parser creates automatically the python

file. Have a look at some of these parsers, especially the write method. Here, the headers, imports, etc. are written. Then the dictionaries are dumped out using `json.dump()`. While this results in a really ugly format for the python file, running `black` over the generated file will properly format everything.

This python file must then be moved to the `iniabu/data` folder. Adjust the `iniabu/data/__init__.py` file to contain imports for the two new dictionaries. Extend the `database_selector()` function with an additional `elif` statement to contain the new database.

Finally, new tests for this database must be added. All tests live in the `test` folder, which has the same structure as the `iniabu` folder that contains the package source code. One good way to write a test is to use an existing test file for a dataset. Then adjust the subroutines and associated asserts. At least make sure that tests exist for:

- Data integrity
- Solar abundance of Si is 10^6
- Relative abundances of all isotopes sum to unity

Finally, add a new test in `test_main.py` to ensure that the database loads correctly. You should add a consistency check for the new database. This ensures that code coverage stays at 100%.

Index

Z