
QCElemental Documentation

Release v0.6.1+0.g444ac8d.dirty

The QCArchive Development Team

Aug 26, 2019

GETTING STARTED

1	Physical Constants	3
2	Periodic Table Data	5
3	Molecule Handlers	7
4	Index	9
	Python Module Index	65
	Index	67

QCElemental is a resource module for quantum chemistry containing physical constants and periodic table data from NIST and molecule handlers.

PHYSICAL CONSTANTS

Physical constants can be acquired directly from the NIST CODATA:

```
>>> qcel.constants.get("hartree energy in ev")  
27.21138602
```

Alternatively, with the use of the Pint unit conversion package, arbitrary conversion factors can be obtained:

```
>>> qcel.constants.conversion_factor("bohr", "miles")  
3.2881547429884475e-14
```


PERIODIC TABLE DATA

A variety of periodic table quantities are available using virtually any alias:

```
>>> qcel.periodictable.to_mass("Ne")
19.9924401762

>>> qcel.periodictable.to_mass(10)
19.9924401762
```


MOLECULE HANDLERS

Molecules can be translated to/from the [MolSSI QCSchema](#) format or quantum chemistry program specific input specifications such as NWChem, Psi4, and CFour. In addition, databases such as PubChem can be searched:

```
>>> qcel.molparse.from_string("pubchem:benzene")
Searching PubChem database for benzene (single best match returned)
Found 1 result(s)
{"geometry": ...}
```


Getting Started

- *Install QCElemental*
- *Community*

4.1 Install QCElemental

You can install `qcelestial` with `conda`, with `pip`, or by installing from source.

4.1.1 Conda

You can update `qcelestial` using `conda`:

```
>>> conda install qcelestial -c conda-forge
```

This installs QCElemental and its dependencies. The `qcelestial` package is maintained on the `conda-forge` channel.

4.1.2 Pip

To install QCElemental with `pip` there are a few options, depending on which dependencies you would like to keep up to date:

```
>>> pip install qcelestial
```

4.1.3 Install from Source

To install QCElemental from source, clone the repository from [github](https://github.com/MolSSI/QCElemental):

```
>>> git clone https://github.com/MolSSI/QCElemental.git
>>> cd QCElemental
>>> python setup.py install
```

or use `pip` for a local install:

```
>>> pip install -e .
```

4.1.4 Testing

QCElemental can be tested using the `pytest` package which can be installed via Conda as well:

```
>>> conda install pytest -c conda-forge
```

Once `pytest` is installed QCElemental's testing suite can be run by:

```
>>> pytest --pyargs qcelemental
```

QCElemental can also be tested from source with:

```
>>> cd QCElemental
>>> pytest
```

4.2 Community

The QCArchive is an open community sponsored by [The Molecular Sciences Software Institute](#). However, this is a community-driven project which requires feature requests, user feedback, and code support. There are a variety of ways to help support the QCArchive project as seen below.

4.2.1 Discussion

- QCArchive Slack is a great place to get feedback and advice from the community. Click [here](#) to get started.
- The QCArchive GitHub repositories contain future roadmaps, current code updates, and a list of issues that are being worked and provide an excellent overview of the development status of the project. See the following links:
 - [QCSchema](#)
 - [QCElemental](#)
 - [QCEngine](#)
 - [QCFractal](#)

4.2.2 Work with us!

The QCArchive project is actively looking for early collaborations to use our tools, help us shake out the bugs, and be evangelists within the computational molecular science community for this code ecosystem. In return you will receive the following benefits:

- Work directly with a MolSSI Software Scientist who will discuss your problem and provide ideas.
- Develop the requirements and potential solutions for your use case within the QCArchive ecosystem.
- Setup monthly meetings to ensure your project stays on track.
- Highlight your project within the QCArchive ecosystem.

If you are interested in working with us, please send an email to QCArchive@molssi.org and we will set up a meeting to discuss specifics.

User Interface

- *Physical Constants*

- [Periodic Table](#)

4.3 Physical Constants

NIST Physical constants are available from QCElemental with arbitrary conversion factors using the [pint](#) package. The current default physical constants come from the [NIST CODATA 2014](#).

4.3.1 Conversion Factors

Conversion factors are available for any valid conversion:

```
>>> qcel.constants.conversion_factor("nanometer", "angstrom")
10.0

>>> qcel.constants.conversion_factor("eV / nanometer ** 2", "hartree / angstrom ** 2")
0.00036749322481535707
```

Warning: QCElemental is explicit: kcal is quite different from kcal / mol. Be careful of common short-hands.

```
>>> qcel.constants.conversion_factor("kcal", "eV")
2.611447418269555e+22

>>> qcel.constants.conversion_factor("kcal / mol", "eV")
0.043364103900593226
```

4.3.2 Quantities

QCElemental supports the [pint](#) “values with units” Quantity objects:

```
>>> q = qcel.constants.Quantity("5 kcal / mol")
>>> q
<Quantity(5, 'kilocalorie')>

>>> q.magnitude
5.0

>>> q.dimensionality
<UnitsContainer({'[length]': 2.0, '[mass]': 1.0, '[substance]': -1.0, '[time]': -2.0})
↪>
```

These objects are often used for code that has many different units to make the requisite bookkeeping nearly effortless. In addition, these objects have NumPy and Pandas support built-in:

```
>>> import numpy as np
>>> a = qcel.constants.Quantity("kcal") * np.arange(4)
>>> a
<Quantity([0 1 2 3], 'kilocalorie')>
```

An example of array manipulation using a NumPy array with a [pint](#) quantity:

```
>>> a * qcel.constants.Quantity("eV")
<Quantity([0 1 2 3], 'electron_volt * kilocalorie')>
```

```
>>> a.to("eV")
<Quantity([0.00000000e+00 2.61144742e+22 5.22289484e+22 7.83434225e+22], 'electron_
↪volt')>
```

4.3.3 NIST CODATA

The exact values from the NIST CODATA can be queried explicitly:

```
>>> qcel.constants.get("hartree energy in ev")
27.21138602
```

The complete NIST CODATA record is held and can be obtained via the Python- API. The following example shows how to obtain a comprehensive overview of the individual CODATA record:

```
>>> datum = qcel.constants.get("hartree energy in ev", return_tuple=True)
>>> datum
<-----
      Datum Hartree energy in eV
-----
Data:      27.21138602
Units:     [eV]
doi:       10.18434/T4WW24
Comment:   uncertainty=0.000 000 17
Glossary:
----->
```

Each of these quantities is API accessible:

```
>>> datum.doi
'10.18434/T4WW24'
>>> datum.comment
'uncertainty=0.000 000 17'
```

4.3.4 Contexts

Physical constants are continuously refined over time as experimental precision increases or redefinition occurs. To prepare for future changes, physical constants are contained in contexts. The `qcel.constants` context will be updated over time to the latest NIST data. To “pin” a context version, a specific context can be created like so:

```
>>> context = qcel.PhysicalConstantsContext("CODATA2014")
>>> context.conversion_factor("hartree", "eV")
27.21138601949571
```

Currently only CODATA2014 is available.

4.3.5 API

Top level user functions:

<code>conversion_factor(base_unit, conv_unit)</code>	Provides the conversion factor from one unit to another.
<code>get(physical_constant[, return_tuple])</code>	Access a physical constant, <i>physical_constant</i> .
<code>Quantity(data)</code>	Returns a Pint Quantity.
<code>string_representation()</code>	Print name, value, and units of all physical constants.

4.3.6 Function Definitions

Note: `conversion_factor` is a function, not a class, but cannot be documented in Sphinx as such due to the way the LRU Cache wraps it. Please disregard the marking of it being a “class.”

class `qcelemental.constants.conversion_factor`

Provides the conversion factor from one unit to another.

The conversion factor is based on the current contexts CODATA.

Parameters

- **base_unit** (`Union[str, 'Quantity']`) – The original units
- **conv_unit** (`Union[str, 'Quantity']`) – The units to convert to

Examples

```
>>> conversion_factor("meter", "picometer")
1e-12
```

```
>>> conversion_factor("feet", "meter")
0.30479999999999996
```

```
>>> conversion_factor(10 * ureg.feet, "meter")
3.0479999999999996
```

Returns The requested conversion factor

Return type `float`

`qcelemental.constants.get(physical_constant: str, return_tuple: bool = False) → Union[float, qcelemental.datum.Datum]`
 Access a physical constant, *physical_constant*.

Parameters

- **physical_constant** (`str`) – Case-insensitive string of physical constant with NIST name.
- **return_tuple** (`bool, optional`) – See below.

Returns When `return_tuple=False`, value of physical constant. When `return_tuple=True`, `Datum` with units, description, uncertainty, and value of physical constant as `Decimal`.

Return type `Union[float, 'Datum']`

`qcelestial.constants.Quantity` (*data: str*) → Quantity
Returns a Pint Quantity.

`qcelestial.constants.string_representation` () → str
Print name, value, and units of all physical constants.

4.4 Periodic Table

Full access to the periodic table is also available within QCElemental. This includes mass, atomic number, symbol, and name information. Data is indexed many ways so that many possible aliases can lead to a single quantity. For example, to obtain the mass (in a.m.u) of Carbon atomic number, symbol, isotope symbol, and name can all be passed:

```
>>> qcel.periodictable.to_mass(6)
12.0
>>> qcel.periodictable.to_mass("C")
12.0
>>> qcel.periodictable.to_mass("C12")
12.0
>>> qcel.periodictable.to_mass("Carbon")
12.0
```

A variety of information can be accessed in this manner. Taking 'Carbon' as an example for mass information:

```
>>> qcel.periodictable.to_mass("Carbon")
12.0
>>> qcel.periodictable.to_mass_number("Carbon")
12

>>> qcel.periodictable.to_atomic_number("Carbon")
6
```

Symbol information:

```
>>> qcel.periodictable.to_symbol("Carbon")
'C'
>>> qcel.periodictable.to_name("Carbon")
'Carbon'
```

Table position information:

```
>>> qcel.periodictable.to_period("Carbon")
2
>>> qcel.periodictable.to_group("Carbon")
14
```

4.4.1 API

Top level user functions:

<code>to_mass(atom, *[, return_decimal])</code>	Get atomic mass of <i>atom</i> .
<code>to_mass_number(atom)</code>	Get mass number of <i>atom</i> .
<code>to_atomic_number(atom)</code>	Get atomic number of <i>atom</i> .

Continued on next page

Table 2 – continued from previous page

<code>to_symbol(atom)</code>	Get element symbol of <i>atom</i> .
<code>to_name(atom)</code>	Get element name of <i>atom</i> .
<code>to_period(atom)</code>	Get period (horizontal row in periodic table) of <i>atom</i> .
<code>to_group(atom)</code>	Get group (vertical column in periodic table) of <i>atom</i> .

4.4.2 Function Definitions

`qcelestial.periodictable.to_mass` (*atom*: Union[int, str], *, *return_decimal*: bool = False) → Union[float, decimal.Decimal]

Get atomic mass of *atom*.

Parameters

- **atom** (*int* or *str*) – Identifier for element or nuclide, e.g., *H*, *D*, *H2*, *He*, *hE4*.
- **return_decimal** (*bool*, *optional*) – Whether to preserve significant figures information by returning as Decimal (*True*) or to convert to float (*False*).

Returns Atomic mass [u]. See above for type. If *atom* is nuclide (e.g., “Li6”), the reported mass. If *atom* is stable element (e.g., “Li”), the mass of the most abundant isotope, “Li7”. If *atom* is unstable element (e.g., “Pu”), the mass of the longest-lived isotope, “Pu244”.

Return type decimal.Decimal or float

Raises *NotAnElementError* – If *atom* cannot be resolved into an element or nuclide.

`qcelestial.periodictable.to_mass_number` (*atom*: Union[int, str]) → int

Get mass number of *atom*.

Functions `to_A()` and `to_mass_number()` are aliases.

Parameters **atom** (*int* or *str*) – Identifier for element or nuclide, e.g., *H*, *D*, *H2*, *He*, *hE4*.

Returns Mass number, number of protons and neutrons. If *atom* is nuclide (e.g., “Li6”), the corresponding mass number, 6. If *atom* is stable element (e.g., “Li”), the mass number of the most abundant isotope, 7. If *atom* is unstable element (e.g., “Pu”), the mass number of the longest-lived isotope, 244.

Return type int

Raises *NotAnElementError* – If *atom* cannot be resolved into an element or nuclide.

`qcelestial.periodictable.to_atomic_number` (*atom*: Union[int, str]) → int

Get atomic number of *atom*.

Functions `to_Z()` and `to_atomic_number()` are aliases.

Parameters **atom** (*int* or *str*) – Identifier for element or nuclide, e.g., *H*, *D*, *H2*, *He*, *hE4*.

Returns Atomic number, number of protons.

Return type int

Raises *NotAnElementError* – If *atom* cannot be resolved into an element or nuclide.

`qcelestial.periodictable.to_symbol` (*atom*: Union[int, str]) → str

Get element symbol of *atom*.

Functions `to_E()` and `to_symbol()` are aliases.

Parameters **atom** (Union[int, str]) – Identifier for element or nuclide, e.g., *H*, *D*, *H2*, *He*, *hE4*.

Returns Element symbol, capitalized.

Return type `str`

`qcelemental.periodictable.to_name` (*atom*: `Union[int, str]`) → `str`
Get element name of *atom*.

Functions `to_element()` and `to_name()` are aliases.

Parameters *atom* (`int` or `str`) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Element name, capitalized.

Return type `str`

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

`qcelemental.periodictable.to_period` (*atom*: `Union[int, str]`) → `int`
Get period (horizontal row in periodic table) of *atom*.

Parameters *atom* (`int` or `str`) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Period between 1 (e.g., *He*) and 7 (e.g., *U238*).

Return type `int`

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

`qcelemental.periodictable.to_group` (*atom*: `Union[int, str]`) → `Optional[int]`
Get group (vertical column in periodic table) of *atom*.

Parameters *atom* (`int` or `str`) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns

- *int* – Group between 1 (e.g., *Li*) and 18 (e.g., *KR84*).
- *None* – If one of the 30 Lanthanides (*Z*=57-71) or Actinides (*Z*=89-103).

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

QCSchema Models

Implementation descriptions of QCSchema objects in Python.

- [Overview](#)
- [Molecule Model](#)

4.5 Overview

Python implementations of the `MolSSI QCSchema` are available within `QCElemental`. These models use `Pydantic` as their base to provide serialization, validation, and manipulation.

4.5.1 Basics

Model creation occurs with a `kwargs` constructor as shown by equivalent operations below:

```
>>> mol = qcel.models.Molecule(symbols=["He"], geometry=[0, 0, 0])
>>> mol = qcel.models.Molecule(**{"symbols": ["He"], "geometry": [0, 0, 0]})
```

A list of all available fields can be found by querying the `fields` attribute:

```
>>> mol.fields.keys()
dict_keys(['symbols', 'geometry', ..., 'id', 'extras'])
```

These attributes can be accessed as shown:

```
>>> mol.symbols
['He']
```

Note that these models are typically immutable:

```
>>> mol.symbols = ["Ne"]
TypeError: "Molecule" is immutable and does not support item assignment
```

To update or alter a model the `copy` command can be used with the `update` kwargs:

```
>>> mol.copy(update={"symbols": ["Ne"]})
< Geometry (in Angstrom), charge = 0.0, multiplicity = 1:

  Center          X          Y          Z
  -----
Ne          0.000000000000    0.000000000000    0.000000000000

>
```

4.5.2 Serialization

All models can be serialized back to their dictionary counterparts through the `dict` function:

```
>>> mol.dict()
{'symbols': ['He'], 'geometry': array([[0., 0., 0.]])}
```

JSON representations are supported out of the box for all models:

```
>>> mol.json()
'{"symbols": ["He"], "geometry": [0.0, 0.0, 0.0]}'
```

Raw JSON can also be parsed back into a model:

```
>>> mol.parse_raw(mol.json())
< Geometry (in Angstrom), charge = 0.0, multiplicity = 1:

  Center          X          Y          Z
  -----
He          0.000000000000    0.000000000000    0.000000000000

>
```

The standard `dict` operation returns all internal representations which may be classes or other complex structures. To return a JSON-like dictionary the `dict` function can be used:

```
>>> mol.dict(encoding='json')
{'symbols': ['He'], 'geometry': [0.0, 0.0, 0.0]}
```

4.6 Molecule Model

A Python implementations of the `MolSSI QCSchema Molecule` object. A “Molecule” many definitions of `Molecule` depending on the domain; this particular `Molecule` is an immutable 3D Cartesian representation with support for quantum chemistry constructs.

4.6.1 Creation

A `Molecule` can be created using the normal kwargs fashion as shown below:

```
>>> mol = qcel.models.Molecule(**{"symbols":["He"], "geometry": [0, 0, 0]})
```

In addition, there is the `from_data` attribute to create a molecule from standard strings:

```
>>> mol = qcel.models.Molecule.from_data("He 0 0 0")
>>> mol
< Geometry (in Angstrom), charge = 0.0, multiplicity = 1:
-----
Center          X          Y          Z
-----
He              0.000000000000  0.000000000000  0.000000000000
>
```

4.6.2 Identifiers

A number of unique identifiers are automatically created for each molecule. Additional implementation such as InChI and SMILES are actively being looked into.

Molecular Hash

A molecule hash is automatically created to allow each molecule to be uniquely identified. The following keys are used to generate the hash:

- `symbols`
- `masses` (1.e-6 tolerance)
- `molecular_charge` (1.e-4 tolerance)
- `molecular_multiplicity`
- `real`
- `geometry` (1.e-8 tolerance)
- `fragments`
- `fragment_charges` (1.e-4 tolerance)
- `fragment_multiplicities`
- `connectivity`

Hashes can be acquired from any molecule object and a `FractalServer` automatically generates canonical hashes when a molecule is added to the database.

```
>>> mol = qcel.models.Molecule(**{"symbols": ["He", "He"], "geometry": [0, 0, -3, 0, 0, 3]})
>>> mol.get_hash()
'84872f975d19aafa62b188b40fbadaf26a3b1f84'
```

Molecular Formula

The molecular formula is also available sorted in alphabetical order with title case symbol names. Any symbol with a count of one does not have a number associated with it.

```
>>> mol.get_molecular_formula()
'He2'
```

4.6.3 Fragments

A Molecule with fragments can be created either using the `--` separators in the `from_data` function or by passing explicit fragments in the `Molecule` constructor:

```
>>> mol = qcel.models.Molecule.from_data(
>>>     """
>>>     Ne 0.000000 0.000000 0.000000
>>>     --
>>>     Ne 3.100000 0.000000 0.000000
>>>     """)

>>> mol = qcel.models.Molecule(
>>>     geometry=[0, 0, 0, 3.1, 0, 0],
>>>     symbols=["Ne", "Ne"],
>>>     fragments=[[0], [1]]
>>> )
```

Fragments from a molecule containing fragment information can be acquired by:

```
>>> mol.get_fragment(0)
< Geometry (in Angstrom), charge = 0.0, multiplicity = 1:

  Center          X          Y          Z
  -----
Ne          0.000000000000    0.000000000000    0.000000000000
>
```

Obtaining fragments with ghost atoms is also supported:

```
>>> mol.get_fragment(0, 1)
< Geometry (in Angstrom), charge = 0.0, multiplicity = 1:

  Center          X          Y          Z
  -----
Ne          0.000000000000    0.000000000000    0.000000000000
Ne (Gh)      3.1000000000572    0.000000000000    0.000000000000
>
```

4.6.4 Molecule Attributes

Below are the documentations for all attributes and initialization keywords for a Molecule object

class `qcelemental.models.Molecule` (*orient=False, validate=None, **kwargs*)

The fundamental representation of a Molecule inside the QCArchive Ecosystem. This model contains data for atomic elements, coordinates, connectivity, charges, fragmentation, etc. The model also supports orientation and measurement utilities.

All Molecule objects are oriented via inertia tensor with the center-of-mass at (0,0,0) unless explicitly blocked.

All Molecule objects have coordinates truncated to 8 (GEOMETRY_NOISE) decimal places and mass and charge truncated to 6 (MASS_NOISE) and 4 (CHARGE_NOISE), respectively, unless explicitly blocked.

Parameters

- **schema_name** (*ConstrainedStrValue, Default: qcschema_molecule*) – The QCSchema specification this model conforms to. Explicitly fixed as `qcschema_molecule`.
- **schema_version** (*int, Default: 2*) – The version number of `schema_name` that this Molecule model conforms to.
- **validated** (*bool, Default: False*) – A boolean indicator (for speed purposes) that the input Molecule data has been previously checked for schema (data layout and type) and physics (e.g., non-overlapping atoms, feasible multiplicity) compliance. This should be `False` in most cases. A `True` setting should only ever be set by the constructor for this class itself or other trusted sources such as a Fractal Server or previously serialized Molecules.
- **symbols** (*Array*) – An ordered (nat,) array-like object of atomic elemental symbols of shape (nat,). The index of this attribute sets atomic order for all other per-atom setting like `real` and the first dimension of `geometry`. Ghost/Virtual atoms must have an entry in this array-like and are indicated by the matching the 0-indexed indices in `real` field.
- **geometry** (*Array*) – An ordered (nat,3) array-like for XYZ atomic coordinates [a0]. Atom ordering is fixed; that is, a consumer who shuffles atoms must not reattach the input (pre-shuffling) molecule schema instance to any output (post-shuffling) per-atom results (e.g., gradient). Index of the first dimension matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. Can also accept array-likes which can be mapped to (nat,3) such as a 1-D list of length $3*\text{nat}$, or the serialized version of the array in $(3*\text{nat},)$ shape; all forms will be reshaped to (nat,3) for this attribute.
- **name** (*str, Optional*) – A common or human-readable name to assign to this molecule. Can be arbitrary.
- **identifiers** (*Identifiers, Optional*) – An optional dictionary of additional identifiers by which this Molecule can be referenced, such as INCHI, canonical SMILES, etc. See the `:class:Identifiers` model for more details.
- **comment** (*str, Optional*) – Additional comments for this Molecule. Intended for pure human/user consumption and clarity.
- **molecular_charge** (*float, Default: 0.0*) – The net electrostatic charge of this Molecule.
- **molecular_multiplicity** (*int, Default: 1*) – The total multiplicity of this Molecule.
- **masses** (*Array, Optional*) – An ordered 1-D array-like object of atomic masses [u] of shape (nat,). Index order matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. If this is not provided, the mass of each atom is inferred from their

most common isotope. If this is provided, it must be the same length as `symbols` but can accept `None` entries for standard masses to infer from the same index in the `symbols` field.

- **real** (*Array, Optional*) – An ordered 1-D array-like object of shape (nat,) indicating if each atom is real (`True`) or ghost/virtual (`False`). Index matches the 0-indexed indices of all other per-atom settings like `symbols` and the first dimension of `geometry`. If this is not provided, all atoms are assumed to be real (`True`). If this is provided, the reality or ghostality of every atom must be specified.
- **atom_labels** (*Array, Optional*) – Additional per-atom labels as a 1-D array-like of strings of shape (nat,). Typical use is in model conversions, such as Elemental \leftrightarrow Molpro and not typically something which should be user assigned. See the `comments` field for general human-consumable text to affix to the Molecule.
- **atomic_numbers** (*Array, Optional*) – An optional ordered 1-D array-like object of atomic numbers of shape (nat,). Index matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. Values are inferred from the `symbols` list if not explicitly set.
- **mass_numbers** (*Array, Optional*) – An optional ordered 1-D array-like object of atomic *mass* numbers of shape (nat). Index matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`. Values are inferred from the most common isotopes of the `symbols` list if not explicitly set.
- **connectivity** (*List[Tuple[int, int, float]], Optional*) – The connectivity information between each atom in the `symbols` array. Each entry in this list is a `Tuple` of (atom_index_A, atom_index_B, bond_order) where the atom_index matches the 0-indexed indices of all other per-atom settings like `symbols` and `real`.
- **fragments** (*List[Array], Optional*) – An indication of which sets of atoms are fragments within the Molecule. This is a list of shape (nfr) of 1-D array-like objects of arbitrary length. Each entry in the list indicates a new fragment. The index of the list matches the 0-indexed indices of `fragment_charges` and `fragment_multiplicities`. The 1-D array-like objects are sets of atom indices indicating the atoms which compose the fragment. The atom indices match the 0-indexed indices of all other per-atom settings like `symbols` and `real`.
- **fragment_charges** (*List[float], Optional*) – The total charge of each fragment in the `fragments` list of shape (nfr,). The index of this list matches the 0-index indices of fragment list. Will be filled in based on a set of rules if not provided (and fragments are specified).
- **fragment_multiplicities** (*List[int], Optional*) – The multiplicity of each fragment in the `fragments` list of shape (nfr,). The index of this list matches the 0-index indices of fragment list. Will be filled in based on a set of rules if not provided (and fragments are specified).
- **fix_com** (*bool, Default: False*) – An indicator which prevents pre-processing the Molecule object to translate the Center-of-Mass to (0,0,0) in euclidean coordinate space. Will result in a different `geometry` than the one provided if `False`.
- **fix_orientation** (*bool, Default: False*) – An indicator which prevents pre-processes the Molecule object to orient via the inertia tensor. Will result in a different `geometry` than the one provided if `False`.
- **fix_symmetry** (*str, Optional*) – Maximal point group symmetry which `geometry` should be treated. Lowercase.

- **provenance** (Provenance, Default: {'creator': 'QCElemental', 'version': 'v0.6.1+0.g444ac8d.dirty', 'routine': 'qcelemental.models.molecule'}) – The provenance information about how this Molecule (and its attributes) were generated, provided, and manipulated.
- **id** (*Any, Optional*) – A unique identifier for this Molecule object. This field exists primarily for Databases (e.g. Fractal’s Server) to track and lookup this specific object and should virtually never need to be manually set.
- **extras** (*Dict[str, Any], Optional*) – Extra information to associate with this Molecule.

Developer Documentation

Contains in-depth developer documentation and API references.

- [QCElemental API](#)
- [Changelog](#)

4.7 QCElemental API

4.7.1 qcelemental Package

Main init for QCElemental

Functions

<code>compare(expected, computed[, label, quiet, ...])</code>	Returns True if two integers, strings, booleans, or integer arrays are element-wise equal.
<code>compare_recursive(expected, computed[, ...])</code>	Recursively compares nested structures such as dictionaries and lists.
<code>compare_values(expected, computed[, label, ...])</code>	Returns True if two floats or float arrays are element-wise equal within a tolerance.

compare

`qcelemental.compare` (*expected, computed, label: str = None, *, quiet: bool = False, return_message: bool = False, return_handler: Callable = None*) → bool

Returns True if two integers, strings, booleans, or integer arrays are element-wise equal.

Parameters

- **expected** (*int, bool, str or int array-like*) – Reference value against which *computed* is compared.
- **computed** (*int, bool, str or int array-like*) – Input value to compare against *expected*.
- **label** (*str, optional*) – Label for passed and error messages. Defaults to calling function name.

Returns

- **allclose** (*bool*) – Returns True if *expected* and *computed* are equal; False otherwise.

- **message** (*str, optional*) – When `return_message=True`, also return passed or error message.

Other Parameters **return_handler** (*function, optional*) – Function to control printing, logging, raising, and returning. Specialized interception for interfacing testing systems.

Notes

- Akin to `np.array_equal`.
- For scalar exactly-comparable types and for arbitrary-dimension, `np.ndarray-castable`, `uniform-type`, exactly-comparable types. For mixed types, use `compare_recursive()`.

compare_recursive

`qcelemental.compare_recursive` (*expected, computed, label: str = None, *, atol: float = 1e-06, rtol: float = 1e-16, forgive: bool = None, quiet: bool = False, return_message: bool = False, return_handler: Callable = None*)
 → bool

Recursively compares nested structures such as dictionaries and lists.

Parameters

- **expected** (*dict*) – Reference value against which *computed* is compared. Dict may be of any depth but should contain Plain Old Data.
- **computed** (*int, bool, str or int array-like*) – Input value to compare against *expected*. Dict may be of any depth but should contain Plain Old Data.
- **atol** (*int or float, optional*) – Absolute tolerance (see formula below).
- **label** (*str, optional*) – Label for passed and error messages. Defaults to calling function name.
- **rtol** (*float, optional*) – Relative tolerance (see formula below). By default set to zero so *atol* dominates.
- **forgive** (*list, optional*) – Keys in top level which may change between *expected* and *computed* without triggering failure.

Returns

- **allclose** (*bool*) – Returns True if *expected* and *computed* are equal within tolerance; False otherwise.
- **message** (*str, optional*) – When `return_message=True`, also return passed or error message.

Notes

```
absolute(computed - expected) <= (atol + rtol * absolute(expected))
```

compare_values

`qcelemental.compare_values` (*expected, computed, label: str = None, *, atol: float = 1e-06, rtol: float = 1e-16, equal_nan: bool = False, passnone: bool = False, quiet: bool = False, return_message: bool = False, return_handler: Callable = None*)
 → bool

Returns True if two floats or float arrays are element-wise equal within a tolerance.

Parameters

- **expected** (*float* or *float array-like*) – Reference value against which *computed* is compared.
- **computed** (*float* or *float array-like*) – Input value to compare against *expected*.
- **atol** (*float*, *optional*) – Absolute tolerance (see formula below).
- **label** (*str*, *optional*) – Label for passed and error messages. Defaults to calling function name.
- **rtol** (*float*, *optional*) – Relative tolerance (see formula below). By default set to zero so *atol* dominates.
- **equal_nan** (*bool*, *optional*) – Passed to `np.isclose`. Compare NaN's as equal.
- **passnone** (*bool*, *optional*) – Return True when both *expected* and *computed* are None.
- **quiet** (*bool*, *optional*) – Whether to log the return message.
- **return_message** (*bool*, *optional*) – Whether to return tuple. See below.

Returns

- **allclose** (*bool*) – Returns True if *expected* and *computed* are equal within tolerance; False otherwise.
- **message** (*str*, *optional*) – When `return_message=True`, also return passed or error message.

Other Parameters **return_handler** (*function*, *optional*) – Function to control printing, logging, raising, and returning. Specialized interception for interfacing testing systems.

Notes

- Akin to `np.allclose`.
- For scalar float-comparable types and for arbitrary-dimension, `np.ndarray-castable`, `uniform-type`, `float-comparable` types. For mixed types, use `compare_recursive()`.
- Sets `rtol` to zero to match expected Psi4 behaviour, otherwise measured as:

```
absolute(computed - expected) <= (atol + rtol * absolute(expected))
```

Classes

<code>ChoicesError(msg[, choices])</code>	Error called for problems with syntax input file.
<code>CovalentRadii([context])</code>	Covalent radii sets.
<code>DataUnavailableError(dataset, atom)</code>	Error when dataset incomplete and otherwise valid query can't be fulfilled.
<code>Datum(label, units, data, *[, comment, doi, ...])</code>	Facilitates the storage of quantum chemical results by labeling them with basic metadata.
<code>MoleculeFormatError(msg)</code>	Error called when a <code>molparse.from_string</code> contains unparsable lines.
<code>NotAnElementError(atom)</code>	Error when element or nuclide can't be identified.

Continued on next page

Table 4 – continued from previous page

<code>PhysicalConstantsContext</code> ([context])	CODATA 2014 physical constants set from NIST.
<code>ValidationError</code> (msg)	Error called for problems with syntax input file.

ChoicesError

exception `qcelemental.ChoicesError` (*msg*, *choices=None*)

Error called for problems with syntax input file. Prints error message *msg* to standard output stream. Also attaches *choices* dictionary with options to proceed.

CovalentRadii

class `qcelemental.CovalentRadii` (*context: str = 'ALVAREZ2008'*)

Bases: `object`

Covalent radii sets.

Parameters `context` (`{ 'ALVAREZ2008' }`) – Origin of loaded data.

cr

Each covalent radius is an entry in *cr*, where key is the “Fe”-cased element symbol if generic or symbol-prefixed label if specialized within element. The value is a Datum object with *lbl* the same as key, *units*, *data* value as Decimal object, and any uncertainty in the *comment* field.

Type `dict` of Datum

doi

The DOI of the current context.

Type `str`

name

The name of the context (`'ALVAREZ2008'`)

Type `str`

native_units

The units the original data was provided in.

Type `str`

year

The year the context was created.

Type `int`

Methods Summary

<code>get</code> (atom, *, <i>return_tuple</i> , <i>units</i> , <i>missing</i>)	Access a covalent radius for species <i>atom</i> .
<code>string_representation</code> ()	Print name, value, and units of all covalent radii.
<code>write_c_header</code> (<i>filename</i> , <i>missing</i>)	Write C header file defining covalent radii array.

Methods Documentation

get (*atom: Union[int, str]*, *, *return_tuple: bool = False*, *units: str = 'bohr'*, *missing: float = None*) → `Union[float, qcelemental.datum.Datum]`
 Access a covalent radius for species *atom*.

Parameters

- **atom** (*int or str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*. In general, one value recommended for each element; however, certain other exact labels may be available. ALVAREZ2008: *C_sp3, C_sp2, C_sp, Mn_lowspin, Mn_highspin, Fe_lowspin, Fe_highspin, Co_lowspin, Co_highspin*
- **units** (*str, optional*) – Units of returned value. To return in native unit (ALVAREZ2008: *angstrom*), pass it explicitly. Only relevant for `return_tuple=False` since `True` returns underlying data structure with native units.
- **missing** (*float or None, optional*) – How to handle when *atom* is valid but outside the available data range. When *None*, raises `DataUnavailableError`. When a float, returns that float, so supply in *units* units. Supplying a float is a more compact assurance that a call will work over all the periodic table than the equivalent “try:
- = `qcel.covalentradii.get(atom) except qcel.DataUnavailableError(rad)` –
- = `4.0``. (rad)` – Only relevant for `return_tuple=False`.
- **return_tuple** (*bool, optional*) – See below.

Returns

- *float* – When `return_tuple=False`, value of covalent radius. If multiple defined for element, returns largest.
- *qcelestial.Datum* – When `return_tuple=True`, `Datum` namedtuple with units, description, uncertainty, and value of covalent radius as `Decimal` (preserving significant figures). If multiple defined for element, returns largest.

Raises

- **NotAnElementError** – If *atom* cannot be resolved into an element or nuclide or label.
- **DataUnavailableError** – If *atom* is a valid element or nuclide but not one for which a covalent radius is available and *missing=None*.

string_representation () → str
Print name, value, and units of all covalent radii.

write_c_header (*filename: str = 'covrad.h', missing: float = 2.0*) → None
Write C header file defining covalent radii array.

Parameters

- **filename** (*str, optional*) – File name for header. Note that changing this won't change the header guard.
- **missing** (*float, optional*) – In order that the C array be atomic-number indexable and that it span the periodic table, this value is used anywhere data is missing.

DataUnavailableError

exception qcelestial.DataUnavailableError (*dataset, atom*)
Error when dataset incomplete and otherwise valid query can't be fulfilled.

Datum

class `qcelemental.Datum`(*label*, *units*, *data*, *, *comment=None*, *doi=None*, *glossary=None*, *numeric=True*)

Bases: `pydantic.main.BaseModel`

Facilitates the storage of quantum chemical results by labeling them with basic metadata.

label

Official label for *data*, often *qcvar*. May contain spaces.

Type `str`

units

ASCII, LaTeX-like representation of units, without square brackets.

Type `str`

data

Value for *label*.

Type `float` or `Decimal` or or `numpy.ndarray`

comment

Additional notes.

Type `str`, optional

doi

Literature citation or definition DOI link.

Type `str`, optional

glossary

Extended description or definition.

Type `str`, optional

numeric

Whether *data* is numeric. Pass *True* to disable validating *data* as `float/Decimal/np.ndarray`.

Type `bool`, optional

MoleculeFormatError

exception `qcelemental.MoleculeFormatError` (*msg*)

Error called when a `molparse.from_string` contains unparsable lines.

NotAnElementError

exception `qcelemental.NotAnElementError` (*atom*)

Error when element or nuclide can't be identified.

PhysicalConstantsContext

class `qcelemental.PhysicalConstantsContext` (*context='CODATA2014'*)

Bases: `object`

CODATA 2014 physical constants set from NIST.

Parameters `context` (`{ 'CODATA2014' }`) – Origin of loaded data.

doi

The DOI of the current context.

Type `str`

name

The name of the context ('CODATA2014')

Type `str`

pc

Each physical constant is an entry in `pc`, where key is the lowercased string of the NIST name (or any alias) and the value is a Datum object with `lbl` the exact NIST name string, `units`, `data` value as Decimal object, and any uncertainty in the `comment` field.

Type dict of Datum

raw_codata

A dictionary representation of the raw context data.

Type Dict[`str`, Any]

year

The year the context was created.

Type `int`

Attributes Summary

<code>conversion_factor</code>	Provides the conversion factor from one unit to another.
<code>ureg</code>	Returns the internal Pint units registry.

Methods Summary

<code>Quantity(data)</code>	Returns a Pint Quantity.
<code>get(physical_constant[, return_tuple])</code>	Access a physical constant, <i>physical_constant</i> .
<code>run_comparison()</code>	Compare the existing physical constant information for Psi4 (in <code>checkup_data</code> folder) to <i>self</i> .
<code>string_representation()</code>	Print name, value, and units of all physical constants.
<code>write_c_header([filename])</code>	Write C header file defining physical constants and pi, all with <code>pc_</code> prefix.

Attributes Documentation

conversion_factor

Provides the conversion factor from one unit to another.

The conversion factor is based on the current contexts CODATA.

Parameters

- **base_unit** (`Union[str, 'Quantity']`) – The original units
- **conv_unit** (`Union[str, 'Quantity']`) – The units to convert to

Examples

```
>>> conversion_factor("meter", "picometer")
1e-12
```

```
>>> conversion_factor("feet", "meter")
0.30479999999999996
```

```
>>> conversion_factor(10 * ureg.feet, "meter")
3.0479999999999996
```

Returns The requested conversion factor

Return type float

ureg

Returns the internal Pint units registry.

Returns The pint context

Return type UnitRegistry

Methods Documentation

Quantity (*data: str*) → Quantity

Returns a Pint Quantity.

get (*physical_constant: str, return_tuple: bool = False*) → Union[float, qcelestial.datum.Datum]

Access a physical constant, *physical_constant*.

Parameters

- **physical_constant** (*str*) – Case-insensitive string of physical constant with NIST name.
- **return_tuple** (*bool, optional*) – See below.

Returns When *return_tuple=False*, value of physical constant. When *return_tuple=True*, Datum with units, description, uncertainty, and value of physical constant as Decimal.

Return type Union[float, 'Datum']

run_comparison () → None

Compare the existing physical constant information for Psi4 (in *checkup_data* folder) to *self*. Specialized use.

string_representation () → str

Print name, value, and units of all physical constants.

write_c_header (*filename='physconst.h'*)

Write C header file defining physical constants and pi, all with *pc_* prefix.

ValidationError

exception qcelestial.ValidationError (*msg*)

Error called for problems with syntax input file. Prints error message *msg* to standard output stream.

Variables

<i>constants</i>	CODATA 2014 physical constants set from NIST.
<i>periodictable</i>	Nuclear and mass data about chemical elements from NIST.

constants

`qcelemental.constants = <qcelemental.physical_constants.context.PhysicalConstantsContext object>`
 CODATA 2014 physical constants set from NIST.

Parameters `context` (`{ 'CODATA2014' }`) – Origin of loaded data.

`qcelemental.doi`
 The DOI of the current context.

Type `str`

`qcelemental.name`
 The name of the context ('CODATA2014')

Type `str`

`qcelemental.pc`
 Each physical constant is an entry in *pc*, where key is the lowercased string of the NIST name (or any alias) and the value is a Datum object with *lbl* the exact NIST name string, *units*, *data* value as Decimal object, and any uncertainty in the *comment* field.

Type `dict of Datum`

`qcelemental.raw_codata`
 A dictionary representation of the raw context data.

Type `Dict[str, Any]`

`qcelemental.year`
 The year the context was created.

Type `int`

periodictable

`qcelemental.periodictable = <qcelemental.periodic_table.PeriodicTable object>`
 Nuclear and mass data about chemical elements from NIST.

Parameters `None` –

`qcelemental.A`
 Mass number, number of protons and neutrons, starting with 0 for dummies.

Type `list of int`

`qcelemental.Z`
 Atomic number, number of protons, starting with 0 for dummies.

Type `list of int`

`qcelemental.E`
 Element symbol from periodic table, starting with "X" for dummies. "Fe" capitalization.

Type list of str

`qcelestial.EA`

Nuclide symbol in E + A form, e.g., “Li6”. List *EA* is a superset of *E*; that is, both “Li6” and “Li” present. For hydrogen, “D” and “T” also included.

Type list of str

`qcelestial.mass`

Atomic mass [u]. For nuclides (e.g., “Li6”), the reported mass. For stable elements (e.g., “Li”), the mass of the most abundant isotope (“Li7”). For unstable elements (e.g., “Pu”), the mass of the longest-lived isotope (“Pu244”).

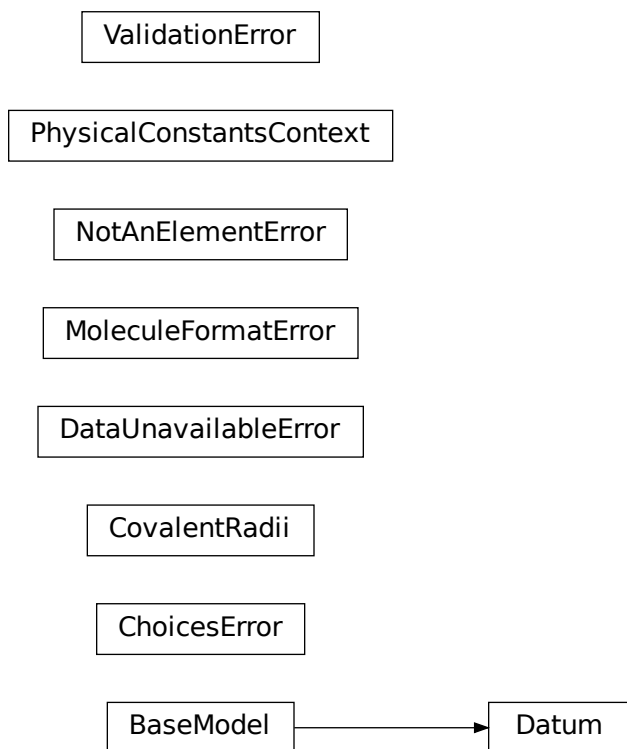
Type list of `decimal.Decimal`

`qcelestial.name`

Element name from periodic table, starting with “Dummy”. “Iron” capitalization.

Type list of str

Class Inheritance Diagram



4.7.2 qcelemental.periodic_table Module

Periodic table class

Classes

<code>PeriodicTable()</code>	Nuclear and mass data about chemical elements from NIST.
------------------------------	--

PeriodicTable

class qcelemental.periodic_table.PeriodicTable

Bases: `object`

Nuclear and mass data about chemical elements from NIST.

Parameters None –

A

Mass number, number of protons and neutrons, starting with 0 for dummies.

Type list of int

Z

Atomic number, number of protons, starting with 0 for dummies.

Type list of int

E

Element symbol from periodic table, starting with “X” for dummies. “Fe” capitalization.

Type list of str

EA

Nuclide symbol in E + A form, e.g., “Li6”. List *EA* is a superset of *E*; that is, both “Li6” and “Li” present. For hydrogen, “D” and “T” also included.

Type list of str

mass

Atomic mass [u]. For nuclides (e.g., “Li6”), the reported mass. For stable elements (e.g., “Li”), the mass of the most abundant isotope (“Li7”). For unstable elements (e.g., “Pu”), the mass of the longest-lived isotope (“Pu244”).

Type list of `decimal.Decimal`

name

Element name from periodic table, starting with “Dummy”. “Iron” capitalization.

Type list of str

Methods Summary

<code>run_comparison()</code>	Compare the existing element information for Psi4 and Cfour (in <code>checkup_data</code> folder) to <i>self</i> .
<code>to_A(atom)</code>	Get mass number of <i>atom</i> .

Continued on next page

Table 10 – continued from previous page

<code>to_E(atom)</code>	Get element symbol of <i>atom</i> .
<code>to_Z(atom)</code>	Get atomic number of <i>atom</i> .
<code>to_atomic_number(atom)</code>	Get atomic number of <i>atom</i> .
<code>to_element(atom)</code>	Get element name of <i>atom</i> .
<code>to_group(atom)</code>	Get group (vertical column in periodic table) of <i>atom</i> .
<code>to_mass(atom, *, return_decimal)</code>	Get atomic mass of <i>atom</i> .
<code>to_mass_number(atom)</code>	Get mass number of <i>atom</i> .
<code>to_name(atom)</code>	Get element name of <i>atom</i> .
<code>to_period(atom)</code>	Get period (horizontal row in periodic table) of <i>atom</i> .
<code>to_symbol(atom)</code>	Get element symbol of <i>atom</i> .
<code>write_c_header([filename])</code>	Write C header file defining arrays of mass and element information.

Methods Documentation

`run_comparison()`

Compare the existing element information for Psi4 and Cfour (in `checkup_data` folder) to *self*. Specialized use.

`to_A(atom: Union[int, str]) → int`

Get mass number of *atom*.

Functions `to_A()` and `to_mass_number()` are aliases.

Parameters `atom` (*int* or *str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Mass number, number of protons and neutrons. If *atom* is nuclide (e.g., “Li6”), the corresponding mass number, 6. If *atom* is stable element (e.g., “Li”), the mass number of the most abundant isotope, 7. If *atom* is unstable element (e.g., “Pu”), the mass number of the longest-lived isotope, 244.

Return type `int`

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

`to_E(atom: Union[int, str]) → str`

Get element symbol of *atom*.

Functions `to_E()` and `to_symbol()` are aliases.

Parameters `atom` (*Union[int, str]*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Element symbol, capitalized.

Return type `str`

`to_Z(atom: Union[int, str]) → int`

Get atomic number of *atom*.

Functions `to_Z()` and `to_atomic_number()` are aliases.

Parameters `atom` (*int* or *str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Atomic number, number of protons.

Return type `int`

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

to_atomic_number (*atom*: Union[int, str]) → int
Get atomic number of *atom*.

Functions *to_Z()* and *to_atomic_number()* are aliases.

Parameters *atom* (*int* or *str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Atomic number, number of protons.

Return type int

Raises *NotAnElementError* – If *atom* cannot be resolved into an element or nuclide.

to_element (*atom*: Union[int, str]) → str
Get element name of *atom*.

Functions *to_element()* and *to_name()* are aliases.

Parameters *atom* (*int* or *str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Element name, capitalized.

Return type str

Raises *NotAnElementError* – If *atom* cannot be resolved into an element or nuclide.

to_group (*atom*: Union[int, str]) → Optional[int]
Get group (vertical column in periodic table) of *atom*.

Parameters *atom* (*int* or *str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns

- *int* – Group between 1 (e.g., *Li*) and 18 (e.g., *KR84*).
- *None* – If one of the 30 Lanthanides (*Z=57-71*) or Actinides (*Z=89-103*).

Raises *NotAnElementError* – If *atom* cannot be resolved into an element or nuclide.

to_mass (*atom*: Union[int, str], *, *return_decimal*: bool = False) → Union[float, decimal.Decimal]
Get atomic mass of *atom*.

Parameters

- *atom* (*int* or *str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.
- *return_decimal* (*bool*, *optional*) – Whether to preserve significant figures information by returning as *Decimal (True)* or to convert to float (*False*).

Returns Atomic mass [u]. See above for type. If *atom* is nuclide (e.g., “*Li6*”), the reported mass. If *atom* is stable element (e.g., “*Li*”), the mass of the most abundant isotope, “*Li7*”. If *atom* is unstable element (e.g., “*Pu*”), the mass of the longest-lived isotope, “*Pu244*”.

Return type decimal.Decimal or float

Raises *NotAnElementError* – If *atom* cannot be resolved into an element or nuclide.

to_mass_number (*atom*: Union[int, str]) → int
Get mass number of *atom*.

Functions *to_A()* and *to_mass_number()* are aliases.

Parameters *atom* (*int* or *str*) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Mass number, number of protons and neutrons. If *atom* is nuclide (e.g., “*Li6*”), the corresponding mass number, 6. If *atom* is stable element (e.g., “*Li*”), the mass number of the most abundant isotope, 7. If *atom* is unstable element (e.g., “*Pu*”), the mass number of the longest-lived isotope, 244.

Return type `int`

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

to_name (*atom*: `Union[int, str]`) → `str`

Get element name of *atom*.

Functions `to_element()` and `to_name()` are aliases.

Parameters *atom* (`int` or `str`) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Element name, capitalized.

Return type `str`

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

to_period (*atom*: `Union[int, str]`) → `int`

Get period (horizontal row in periodic table) of *atom*.

Parameters *atom* (`int` or `str`) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Period between 1 (e.g., *He*) and 7 (e.g., *U238*).

Return type `int`

Raises `NotAnElementError` – If *atom* cannot be resolved into an element or nuclide.

to_symbol (*atom*: `Union[int, str]`) → `str`

Get element symbol of *atom*.

Functions `to_E()` and `to_symbol()` are aliases.

Parameters *atom* (`Union[int, str]`) – Identifier for element or nuclide, e.g., *H, D, H2, He, hE4*.

Returns Element symbol, capitalized.

Return type `str`

write_c_header (*filename*: `str = 'masses.h'`) → `None`

Write C header file defining arrays of mass and element information.

Parameters *filename* (`str`, *optional*) – The filename to write to.

Class Inheritance Diagram

PeriodicTable

4.7.3 qcelemental.physical_constants Package

Units handlers

Classes

<i>PhysicalConstantsContext</i> ([context])	CODATA 2014 physical constants set from NIST.
---	---

PhysicalConstantsContext

class qcelestial.physical_constants.**PhysicalConstantsContext** (*context*='CODATA2014')

Bases: `object`

CODATA 2014 physical constants set from NIST.

Parameters **context** (`{ 'CODATA2014' }`) – Origin of loaded data.

doi

The DOI of the current context.

Type `str`

name

The name of the context ('CODATA2014')

Type `str`

pc

Each physical constant is an entry in *pc*, where key is the lowercased string of the NIST name (or any alias) and the value is a Datum object with *lbl* the exact NIST name string, *units*, *data* value as Decimal object, and any uncertainty in the *comment* field.

Type `dict of Datum`

raw_codata

A dictionary representation of the raw context data.

Type `Dict[str, Any]`

year

The year the context was created.

Type `int`

Attributes Summary

<i>conversion_factor</i>	Provides the conversion factor from one unit to another.
<i>ureg</i>	Returns the internal Pint units registry.

Methods Summary

<i>Quantity</i> (data)	Returns a Pint Quantity.
<i>get</i> (physical_constant[, return_tuple])	Access a physical constant, <i>physical_constant</i> .
<i>run_comparison</i> ()	Compare the existing physical constant information for Psi4 (in <i>checkup_data</i> folder) to <i>self</i> .
<i>string_representation</i> ()	Print name, value, and units of all physical constants.
<i>write_c_header</i> ([filename])	Write C header file defining physical constants and pi, all with <code>pc_</code> prefix.

Attributes Documentation

conversion_factor

Provides the conversion factor from one unit to another.

The conversion factor is based on the current contexts CODATA.

Parameters

- **base_unit** (*Union[str, 'Quantity']*) – The original units
- **conv_unit** (*Union[str, 'Quantity']*) – The units to convert to

Examples

```
>>> conversion_factor("meter", "picometer")
1e-12
```

```
>>> conversion_factor("feet", "meter")
0.30479999999999996
```

```
>>> conversion_factor(10 * ureg.feet, "meter")
3.0479999999999996
```

Returns The requested conversion factor

Return type float

ureg

Returns the internal Pint units registry.

Returns The pint context

Return type UnitRegistry

Methods Documentation

Quantity (*data: str*) → Quantity

Returns a Pint Quantity.

get (*physical_constant: str, return_tuple: bool = False*) → Union[float, qcelestial.datum.Datum]

Access a physical constant, *physical_constant*.

Parameters

- **physical_constant** (*str*) – Case-insensitive string of physical constant with NIST name.
- **return_tuple** (*bool, optional*) – See below.

Returns When *return_tuple=False*, value of physical constant. When *return_tuple=True*, Datum with units, description, uncertainty, and value of physical constant as Decimal.

Return type Union[float, 'Datum']

run_comparison () → None

Compare the existing physical constant information for Psi4 (in *checkup_data* folder) to *self*. Specialized use.

string_representation () → str
Print name, value, and units of all physical constants.

write_c_header (*filename='physconst.h'*)
Write C header file defining physical constants and pi, all with `pc_` prefix.

Class Inheritance Diagram

```

classDiagram
    class PhysicalConstantsContext
  
```

4.7.4 qcelemental.molparse Package

Functions

<code>contiguize_from_fragment_pattern</code> (<i>frag_pattern</i> *, *)	Take (nat, ?) array-like arrays and return with atoms arranged by (nfr, ?) <i>frag_pattern</i> .
<code>from_arrays</code> (*[, <i>geom</i> , <i>elea</i> , <i>elez</i> , <i>elem</i> , ...])	Compose a Molecule dict from unvalidated arrays and variables, returning dict.
<code>from_input_arrays</code> (*[, <i>enable_qm</i> , ...])	Compose a Molecule dict from unvalidated arrays and variables in multiple domains.
<code>from_schema</code> (<i>molschema</i> , *[, <i>verbose</i>])	Construct molecule dictionary representation from non-Psi4 schema.
<code>from_string</code> (<i>molstr</i> [, <i>dtype</i> , <i>name</i> , <i>fix_com</i> , ...])	Construct a molecule dictionary from any recognized string format.
<code>parse_nucleus_label</code> (<i>label</i>)	Separate molecule nucleus string into fields.
<code>reconcile_nucleus</code>	Forms consistent set of nucleus descriptors from all information from arguments, supplemented by the periodic table.
<code>to_schema</code> (<i>molrec</i> , <i>dtype</i> [, <i>units</i> , <i>np_out</i> , <i>copy</i>])	Translate molparse internal Molecule spec into dictionary from other schemas.
<code>to_string</code> (<i>molrec</i> , <i>dtype</i> [, <i>units</i> , ...])	Format a string representation of QM molecule.
<code>validate_and_fill_chgmult</code> (<i>zeff</i> , ...[, ...])	Forms molecular and fragment charge and multiplicity specification by completing and reconciling information from argument, supplemented by physical constraints and sensible defaults.

contiguize_from_fragment_pattern

`qcelemental.molparse.contiguize_from_fragment_pattern` (*frag_pattern*, *, *geom=None*,
verbose: int = 1,
throw_reorder: bool = False, ***kwargs*)

Take (nat, ?) array-like arrays and return with atoms arranged by (nfr, ?) *frag_pattern*.

Parameters

- **frag_pattern** (*list of lists of ints*) – (nfr, ?) list of indices (0-indexed) grouping atoms into molecular fragments within the topology.
- **geom** (*array-like, optional*) – (nat, 3) or (3 * nat,) ndarray or list o’lists of Cartesian coordinates, possibly with atoms belonging to the same fragment being dispersed in *geom*.
- **throw_reorder** (*bool, optional*) – Whether, when non-contiguous fragments detected, to raise `ValidationError` (`True`) or to proceed to reorder atoms to contiguize fragments (`False`).
- **verbose** (*int, optional*) – Quantity of printing
- **kwargs** (*None or array-like*) – Each additional array will be returned with ordering applied in the return dictionary.

Returns

- **fragment_separators** (*array-like of int*) – (nfr - 1,) list of atom indices at which to split *geom* into fragments.
- **geom** (*ndarray of float, optional*) – (3 * nat,) Cartesian coordinates with fragments contiguous.
- **kwargs** (*None or ndarray, optional*) – (nat,) Each *kwargs* input array reordered for contiguous fragments.

Raises `qcelestial.ValidationError` – When *frag_pattern* skips atoms or any array has inconsistent length. If *throw_reorder*, raises when non-contiguous fragments detected.

from_arrays

```
qcelestial.molparse.from_arrays(*, geom=None, elea=None, elez=None, elem=None,
                                mass=None, real=None, elbl=None, name=None,
                                units='Angstrom', input_units_to_au=None, fix_com=None,
                                fix_orientation=None, fix_symmetry=None, frag-
                                ment_separators=None, fragment_charges=None, frag-
                                ment_multiplicities=None, molecular_charge=None, molec-
                                ular_multiplicity=None, comment=None, provenance=None,
                                connectivity=None, fragment_files=None, hint_types=None,
                                geom_hints=None, geom_unsettled=None, variables=None,
                                domain='qm', missing_enabled_return='error',
                                np_out=True, speclabel=True, tooclose=0.1,
                                zero_ghost_fragments=False, nonphysical=False,
                                mtol=0.001, copy=True, verbose=1)
```

Compose a Molecule dict from unvalidated arrays and variables, returning dict.

See fields of Return molrec below. Required parameters (for QM XYZ) are *geom* and one of *elem*, *elez*, *elbl* (*speclabel=True*)

Parameters

- **geom** (*array-like*) – (nat, 3) or (3 * nat,) ndarray or list o’lists of Cartesian coordinates.
- **fragment_separators** (*array-like of int, optional*) – (nfr - 1,) list of atom indices at which to split *geom* into fragments.
- **elbl** (*ndarray of str*) – (nat,) Label extending *elem* symbol, possibly conveying ghosting, isotope, mass, tagging information.

- **tooclose** (*float, optional*) – Interatom distance (native *geom* units) nearer than which atoms not allowed.
- **nonphysical** (*bool, optional*) –
- **speclabel** (*bool, optional*) – If *True*, interpret *elbl* as potentially full nucleus spec including ghosting, isotope, mass, tagging information, e.g., *@13C_mine* or *He4@4.01*. If *False*, interpret *elbl* as only the user/tagging extension to nucleus label, e.g. *_mine* or *4* in the previous examples.
- **missing_enabled_return** (*{'minimal', 'none', 'error'}*) – What to do when an enabled domain is of zero-length? Respectively, return a fully valid but empty molrec, return empty dictionary, or throw error.
- **np_out** (*bool, optional*) – When *True*, fields *geom, elea, elez, elem, mass, real, elbl* will be ndarray. Use *False* to get a json-able version.

Returns

- **molrec** (*dict*) – Molecule dictionary spec follows. Its principles are
 - (1) contents are fully validated and defaulted - no error checking necessary,
 - (2) contents may be mildly redundant - atomic numbers and element symbols present,
 - (3) big system, nat-length single-type arrays, not small system, nat-number heterogeneous objects,
 - (4) some fields are optional (e.g., *fix_symmetry*) but largely self-describing so units or *fix_com* must be present.
 - (5) apart from some mild optional fields, *_all_* fields will be present (corollary of “fully validated and defaulted”) - no need to check for every key. in some cases like *efp*, keys will appear in blocks, so pre-handshake there will be a few hint keys and post-handshake they will be joined by full qm-like molrec.
 - (6) molrec should be idempotent through this function (equiv to schema validator) but are not idempotent throughout its life. if fields permit, frame may be changed. Future? if fields permit, mol may be symmetrized. Coordinates and angles may change units or range if program returns them in only one form.
- **name** (*str, optional*) – Label for molecule; should be valid Python identifier.
- **units** (*{'Angstrom', 'Bohr'}*) – Units for *geom*.
- **input_units_to_au** (*float, optional*) – If *units='Angstrom'*, overrides consumer’s value for [A]->[a0] conversion.
- **fix_com** (*bool*) – Whether translation of *geom* is allowed or disallowed.
- **fix_orientation** (*bool*) – Whether rotation of *geom* is allowed or disallowed.
- **fix_symmetry** (*str, optional*) – Maximal point group symmetry which *geom* should be treated. Lowercase.
- **geom** (*ndarray of float*) – (3 * nat,) Cartesian coordinates in *units*.
- **elea** (*ndarray of int*) – (nat,) Mass number for atoms, if known isotope, else -1.
- **elez** (*ndarray of int*) – (nat,) Number of protons, nuclear charge for atoms.
- **elem** (*ndarray of str*) – (nat,) Element symbol for atoms.
- **mass** (*ndarray of float*) – (nat,) Atomic mass [u] for atoms.
- **real** (*ndarray of bool*) – (nat,) Real/ghostedness for atoms.

- **elbl** (*ndarray of str*) – (nat,) Label with any tagging information from element spec.
- **fragment_separators** (*list of int*) – (nfr - 1,) list of atom indices at which to split *geom* into fragments.
- **fragment_charges** (*list of float*) – (nfr,) list of charge allocated to each fragment.
- **fragment_multiplicities** (*list of int*) – (nfr,) list of multiplicity allocated to each fragment.
- **molecular_charge** (*float*) – total charge on system.
- **molecular_multiplicity** (*int*) – total multiplicity on system.
- **comment** (*str; optional*) – Additional comment for molecule.
- **provenance** (*dict of str*) – Accumulated history of molecule, with fields “creator”, “version”, “routine”.
- **connectivity** (*list of tuples of int, optional*) – (nbond, 3) list of (0-indexed) (atomA, atomB, bond_order) (int, int, double) tuples
- *EFP extension (this + units is minimal)*
- **fragment_files** (*list of str*) – (nfr,) lowercased names of efp meat fragment files.
- **hint_types** (*{'xyzabc', 'points'}*) – (nfr,) type of fragment orientation hint.
- **geom_hints** (*list of lists of float*) – (nfr,) inner lists have length 6 (xyzabc; to orient the center) or 9 (points; to orient the first three atoms) of the EFP fragment.
- *QMVZ extension (geom_unsettled replaces geom)*
- **geom_unsettled** (*list of lists of str*) – (nat,) all-string Cartesian and/or zmat anchor and value contents mixing anchors, values, and variables.
- **variables** (*list of pairs*) – (nvar, 2) pairs of variables (str) and values (float). May be incomplete.

Raises `qcelestial.ValidationError` – For most anything wrong.

from_input_arrays

```
qcelestial.molparse.from_input_arrays(*,
                                     enable_qm=True,      enable_efp=True,
                                     missing_enabled_return_qm='error',
                                     missing_enabled_return_efp='error',
                                     geom=None,
                                     elea=None,           elez=None,           elem=None,
                                     mass=None,           real=None,           elbl=None,
                                     name=None,           units='Angstrom',    input_units_to_au=None,
                                     fix_com=None,
                                     fix_orientation=None, fix_symmetry=None,
                                     fragment_separators=None,
                                     fragment_charges=None,
                                     fragment_multiplicities=None,
                                     molecular_charge=None,
                                     molecular_multiplicity=None,
                                     fragment_files=None,
                                     hint_types=None,
                                     geom_hints=None,
                                     geom_unsettled=None,
                                     variables=None,
                                     speclabel=True,
                                     tooclose=0.1,
                                     zero_ghost_fragments=False,
                                     nonphysical=False,
                                     mtol=0.001,
                                     copy=True,
                                     verbose=1)
```

Compose a Molecule dict from unvalidated arrays and variables in multiple domains.

Drives `qcelestial.molparse.from_arrays()` for successive domains and hooks them together (e.g., impose *fix_com* on “qm” when “efp” present).

from_schema

`qcelestial.molparse.from_schema(molschema, *, verbose: int = 1) → Dict`
Construct molecule dictionary representation from non-Psi4 schema.

Parameters

- **molschema** (*dict*) – Dictionary form of Molecule following known schema.
- **verbose** (*int, optional*) – Amount of printing.

Returns `molrec` – Dictionary representation of instance.

Return type `dict`

from_string

`qcelestial.molparse.from_string(molstr, dtype=None, *, name=None, fix_com=None, fix_orientation=None, fix_symmetry=None, return_processed=False, enable_qm=True, enable_efp=True, missing_enabled_return_qm='none', missing_enabled_return_efp='none', verbose=1) → Union[Dict, Tuple[Dict, Dict]]`

Construct a molecule dictionary from any recognized string format.

Parameters

- **molstr** (*str*) – Multiline string specification of molecule in a recognized format.
- **dtype** (*{'xyz', 'xyz+', 'psi4', 'psi4+'}, optional*) – Molecule format name; see below for details.
- **return_processed** (*bool, optional*) – Additionally return intermediate dictionary.
- **enable_qm** (*bool, optional*) – Consider quantum mechanical domain in processing the string constants into the returned molrec.
- **enable_efp** (*bool, optional*) – Consider effective fragment potential domain in processing the string contents into the returned molrec. Only relevant if *dtype* supports EFP.
- **missing_enabled_return_qm** (*{'minimal', 'none', 'error'}*) – If *enable_qm=True*, what to do if it has no atoms/fragments? Respectively, return a fully valid but empty molrec, return empty dictionary, or throw error.
- **missing_enabled_return_efp** (*{'minimal', 'none', 'error'}*) – If *enable_efp=True*, what to do if it has no atoms/fragments? Respectively, return a fully valid but empty molrec, return empty dictionary, or throw error.
- **name** (*str, optional*) – Override *molstr* information for label for molecule; should be valid Python identifier. One of a very limited number of fields (three others follow) for trumping *molstr*. Provided for convenience, since the alternative would be collect the resulting molrec (discarding the Mol if called from class), editing it, then remaking the Mol.
- **fix_com** (*bool, optional*) – Override *molstr* information for whether translation of *geom* is allowed or disallowed.
- **fix_orientation** (*bool, optional*) – Override *molstr* information for whether rotation of *geom* is allowed or disallowed.

- **fix_symmetry** (*str*, *optional*) – Override *molstr* information for maximal point group symmetry which geometry should be treated.

Returns

- **molrec** (*dict*) – Molecule dictionary spec. See `from_arrays()`.
- **molinit** (*dict*, *optional*) – Intermediate “molrec”-like dictionary containing *molstr* info after parsing by this function but before the validation and defaulting of `from_arrays` that returns the proper *molrec*. Only provided if `return_processed` is True.

Raises `qcelemental.MoleculeFormatError` – After processing of *molstr*, only an empty string should remain. Anything left is a syntax error.

Notes

Several formats are interpretable

```

xyz - Strict XYZ format
-----

String Layout
-----
<number of atoms>
comment line
<element_symbol or atomic_number> <x> <y> <z>
...
<element_symbol or atomic_number> <x> <y> <z>

QM Domain
-----
Specifiabile: geom, elem/elez (element identity)
Inaccessible: mass, real (vs. ghost), elbl (user label), name, units (assumed_
↪[A]),
                input_units_to_au, fix_com/orientation/symmetry, fragmentation,
                molecular_charge, molecular_multiplicity

Notes
-----
<number of atoms> is pattern-matched but ignored.

xyz+ - Enhanced XYZ format
-----

String Layout
-----
<number of atoms> [<bohr|au|ang>]
[<molecular_charge> <molecular_multiplicity>] comment line
<psi4_nucleus_spec> <x> <y> <z>
...
<psi4_nucleus_spec> <x> <y> <z>

QM Domain
-----
Specifiabile: geom, elem/elez (element identity), mass, real (vs. ghost), elbl_
↪(user label),
                units (defaults [A]), molecular_charge, molecular_multiplicity
Inaccessible: name, input_units_to_au, fix_com/orientation/symmetry, ↪
↪fragmentation
    
```

(continues on next page)

(continued from previous page)

```

Notes
-----
<number of atoms> is pattern-matched but ignored.

psi4 - Psi4 molecule {...} format
-----

QM Domain
-----
Specifiable: geom, elem/elez (element identity), mass, real (vs. ghost), elbl_
↳(user label),
           units (defaults [A]), fix_com/orientation/symmetry, fragment_
↳separators,
           fragment_charges, fragment_multiplicities, molecular_charge,
↳molecular_multiplicity
Inaccessible: name, input_units_to_au

PubChem
-----
pubchem : <cid|name|formula> [*]

A string like the above searches the PubChem database and substitutes the_
↳below. Adding the wildcard
           searches for multiple matches and raises ChoicesError with matches for_
↳further consideration attached.

Specifiable: geom, elem/elez (element identity), units (fixed [A]),_
↳molecular_charge,
           molecular_multiplicity (fixed singlet), name

EFP Domain
-----
Specifiable: units, fix_com/orientation/symmetry, fragment_files, hint_types,
↳geom_hints
Inaccessible: anything atomic or fragment details -- geom, elem/elez (element_
↳identity),
           mass, real (vs. ghost), elbl (user label), fragment_separators,
↳fragment_charges,
           fragment_multiplicities, molecular_charge, molecular_
↳multiplicity

psi4+ - Psi4 non-Cartesian molecule {...} format
-----
Like `dtype=psi4` (although combination with EFP not tested) except
that instead of pure-Cartesian geometry, allow variables, zmatrix,
and un-fully-specified geometries. *Not* MolSSI standard, but we're
not dropping zmatrix yet. Note that in Psi4 internal coordinates
defined through a zmatrix have no bearing on geometry
optimization internals or constraints.

```

parse_nucleus_label

qcelestial.molparse.parse_nucleus_label(*label*)

Separate molecule nucleus string into fields.

Parameters `label` (*str*) – Conveys at least element and ghostedness and possibly isotope, mass, and user info in accordance with `qcelemental.molparse.regex.NUCLEUS`.

Returns `A, Z, E, mass, real, user` – Field breakdown of `label`.

Return type `int` or `None`, `int` or `None`, `str` or `None`, `float` or `None`, `bool`, `str` or `None`

Raises `qcelemental.ValidationError` – If `label` does not match `NUCLEUS`.

Examples

```
>>> parse_nucleus_label('@ca_miNe')
None, None, 'ca', None False, '_miNe'
```

```
>>> parse_nucleus_label('Gh(Ca_mine)')
None, None, 'Ca', None '_mine', False
```

```
>>> parse_nucleus_label('@Ca_mine@1.07')
None, None, 'Ca', 1.07 False, '_mine'
```

```
>>> parse_nucleus_label('Gh(cA_MINE@1.07)')
None, None, 'cA', 1.07 False, '_MINE'
```

```
>>> parse_nucleus_label('@40Ca_mine@1.07')
40, None, 'Ca', 1.07 False, '_mine'
```

```
>>> parse_nucleus_label('Gh(40Ca_mine@1.07)')
40, None, 'Ca', 1.07 False, '_mine'
```

```
>>> parse_nucleus_label('444lu333@4.0')
444, None, 'lu', 4.0 True, '333'
```

```
>>> parse_nucleus_label('@444lu333@4.4')
444, None, 'lu', 4.4 False, '333'
```

```
>>> parse_nucleus_label('8i')
8, None, 'i', None True, None
```

```
>>> parse_nucleus_label('53_mI4')
None, 53, None, None True, '_mI4'
```

```
>>> parse_nucleus_label('@5_MINEs3@4.4')
None, 5, None, 4.4 False, '_MINEs3'
```

```
>>> parse_nucleus_label('Gh(555_mines3@0.1)')
None, 555, None, 0.1 False, '_mines3'
```

molparse.reconcile_nucleus

molparse.reconcile_nucleus

Forms consistent set of nucleus descriptors from all information from arguments, supplemented by the periodic table. At the least, must provide element identity somehow. Defaults to most-abundant isotope.

Parameters

- **A** (*int, optional*) – Mass number, number of protons and neutrons.
- **Z** (*int, optional*) – Atomic number, number of protons.
- **E** (*str, optional*) – Element symbol from periodic table.
- **mass** (*float, optional*) – Atomic mass [u].
- **real** (*bool, optional*) – Whether real or ghost/absent.
- **label** (*str, optional*) – Atom label according to `qcelemental.molparse.regex.NUCLEUS`.
- **speclabel** (*bool, optional*) – If *True*, interpret *label* as potentially full nucleus spec including ghosting, isotope, mass, tagging information, e.g., `@13C_mine` or `He4@4.01`. If *False*, interpret *label* as only the user/tagging extension to nucleus label, e.g. `_mine` or `4` in the previous examples.
- **nonphysical** (*bool, optional*) – When *True*, turns off sanity checking that prevents periodic table violations (e.g. light uranium: `1U@1.007`).
- **mtol** (*float, optional*) – How different *mass* can be from a known nuclide mass and still merit the mass number assignment. Note that for elements dominated by a single isotope, the default may not be tight enough to prevent standard atomic weight (abundance-average of isotopes) from being labeled as the dominant isotope for A.
- **verbose** (*int, optional*) – Quantity of printing.

Returns **A, Z, E, mass, real, userlabel** – mass number, unless clues don't point to a known nuclide, in which case *-1*. atomic number. element symbol, capitalized. mass value [u]. real/ghost. user portion of *label* if present, else ''.

Return type `int, int, str, float, bool, str`

Raises

- `qcelemental.NotAnElementError` –
- `qcelemental.ValidationError` –

Examples

```
>>> reconcile_nucleus(E='co')
>>> reconcile_nucleus(Z=27)
>>> reconcile_nucleus(A=59, Z=27)
>>> reconcile_nucleus(E='cO', mass=58.933195048)
>>> reconcile_nucleus(A=59, Z=27, E='CO')
>>> reconcile_nucleus(A=59, E='cO', mass=58.933195048)
>>> reconcile_nucleus(label='co')
>>> reconcile_nucleus(label='59co')
>>> reconcile_nucleus(label='co@58.933195048')
>>> reconcile_nucleus(A=59, Z=27, E='cO', mass=58.933195048, label='co@58.
↪933195048')
>>> reconcile_nucleus(A=59, Z=27, E='cO', mass=58.933195048, label='27@58.
↪933195048')
>>> reconcile_nucleus(label='27')
59, 27, 'Co', 58.933195048, True, ''
```

```
>>> reconcile_nucleus(label='co_miNe')
>>> reconcile_nucleus(label='co_mIne@58.933195048')
59, 27, 'Co', 58.933195048, True, '_mine'
```

```
>>> reconcile_nucleus(E='cO', mass=58.933)
>>> reconcile_nucleus(label='cO@58.933')
59, 27, 'Co', 58.933, True, ''
>>> assert 59, 27, 'Co', 58.933, True, '' == reconcile_nucleus(E='cO', mass=58.
↳933, mtol=1.e-4))
AssertionError
```

```
>>> reconcile_nucleus(E='Co', A=60)
>>> reconcile_nucleus(Z=27, A=60, real=True)
>>> reconcile_nucleus(E='Co', A=60)
>>> reconcile_nucleus(Z=27, mass=59.933817059)
>>> reconcile_nucleus(A=60, Z=27, mass=59.933817059)
>>> reconcile_nucleus(label='60Co')
>>> reconcile_nucleus(label='27', mass=59.933817059)
>>> reconcile_nucleus(label='Co', mass=59.933817059)
>>> reconcile_nucleus(A=60, label='Co')
60, 27, 'Co', 59.933817059, True, ''
```

```
>>> reconcile_nucleus(E='Co', A=60, real=False)
>>> reconcile_nucleus(A=60, Z=27, mass=59.933817059, real=0)
>>> reconcile_nucleus(label='@60Co')
>>> reconcile_nucleus(label='Gh(27)', mass=59.933817059)
>>> reconcile_nucleus(label='@Co', mass=59.933817059)
>>> reconcile_nucleus(A=60, label='Gh(Co)')
60, 27, 'Co', 59.933817059, False, ''
```

```
>>> reconcile_nucleus(Z=27, mass=200, nonphysical=True)
60, 27, 'Co', 200.00000000, True, ''
```

```
>>> reconcile_nucleus(mass=60.6, Z=27)
>>> reconcile_nucleus(mass=60.6, E='Co')
>>> reconcile_nucleus(mass=60.6, label='27')
>>> reconcile_nucleus(label='Co@60.6')
-1, 27, 'Co', 60.6, True, ''
```

```
>>> reconcile_nucleus(mass=60.6, Z=27, A=61)
>>> reconcile_nucleus(A=80, Z=27)
>>> reconcile_nucleus(Z=27, mass=200)
>>> reconcile_nucleus(Z=27, mass=-200, nonphysical=True)
>>> reconcile_nucleus(Z=-27, mass=200, nonphysical=True)
>>> reconcile_nucleus(Z=1, label='he')
>>> reconcile_nucleus(A=4, label='3he')
>>> reconcile_nucleus(label='@U', real=True)
>>> reconcile_nucleus(label='U', real=False)
ValidationError
```

to_schema

`qcelemental.molparse.to_schema` (*molrec*: Dict[str, Any], *dtype*: str, *units*: str = 'Bohr', *, *np_out*: bool = False, *copy*: bool = True) → Dict[str, Any]

Translate molparse internal Molecule spec into dictionary from other schemas.

Parameters

- **molrec** (*dict*) – Psi4 json Molecule spec.
- **dtype** ({'psi4', 1, 2}) – Molecule schema format. 1 is https://molssi-qc-schema.readthedocs.io/en/latest/auto_topology.html V1 + #44 + #53 2 is 1 with internal schema_name/version (<https://github.com/MolSSI/QCSchema/pull/60>)
- **units** ({'Bohr', 'Angstrom'}) – Units in which to write string. There is not an option to write in intrinsic/input units. Some *dtype* may not allow all units.
- **np_out** (*bool*, *optional*) – When *True*, fields originating from *geom*, *elea*, *elez*, *elem*, *mass*, *real*, *elbl* will be ndarray. Use *False* to get a json-able version.
- **#return_type** ({'json', 'yaml'}) *Serialization format string to return.* –

Returns `qcschema` – Dictionary of the *dtype* repr of *molrec*.

Return type `dict`

to_string

`qcelemental.molparse.to_string` (*molrec*: Dict, *dtype*: str, *units*: str = None, *, *atom_format*: str = None, *ghost_format*: str = None, *width*: int = 17, *prec*: int = 12, *return_data*: bool = False)

Format a string representation of QM molecule.

Parameters

- **molrec** (*dict*) – Psi4 json Molecule spec.
- **dtype** ({'xyz', 'cfour', 'nwchem', 'molpro'}) – Overall string format. Note that it's possible to request variations that don't fit the *dtype* spec so may not be readable (e.g., *ghost* and *mass* in nucleus label with 'xyz'). 'cfour' forces nucleus label, ignoring *atom_format*, *ghost_format*
- **units** (*str*, *optional*) – Units in which to write string. Usually *Angstrom* or *Bohr* but may be any length unit. There is not an option to write in intrinsic/input units. For *dtype*='xyz', *units*='Bohr' where the format doesn't have a slot to specify units, "au" is added so that readable as *dtype*='xyz+'.
- **atom_format** (*str*, *optional*) – General format is '{elem}'. A format string that may contain fields 'elea' (-1 will be ''), 'elez', 'elem', 'mass', 'elbl' in any arrangement. For example, if a format naturally uses element symbol and you want atomic number instead with mass info, too, pass '{elez}@{mass}'. See *ghost_format* for handling field 'real'.
- **ghost_format** (*str*, *optional*) – General format is '@{elem}'. Like *atom_format*, but this formatter is used when *real*=False. To suppress ghost atoms, use *ghost_format*="".
- **width** (*int*, *optional*) – Field width for formatting coordinate float.
- **prec** (*int*, *optional*) – Number of decimal places for formatting coordinate float.

- **data** (*return*) – Whether to return dictionary with additional info from the molrec that’s not expressible in the string but may be of interest to the QC program. Note that field names are in QCSchema, not molrec, language.

Returns

- *str* – String representation of the molecule.
- *str, dict* –

When “**return_data=True**”, return additionally a dictionary keywords: key, value pairs for processing molecule info into options fields: aspects of `qcelestial.models.Molecule` expressed into `string_or_` keywords.

validate_and_fill_chgmult

`qcelestial.molparse.validate_and_fill_chgmult` (*zeff*, *fragment_separators*, *molecular_charge*: *Optional[float]*, *fragment_charges*, *molecular_multiplicity*: *Optional[int]*, *fragment_multiplicities*, *zero_ghost_fragments*: *bool = False*, *verbose*: *int = 1*) → Dict[str, Any]

Forms molecular and fragment charge and multiplicity specification by completing and reconciling information from argument, supplemented by physical constraints and sensible defaults.

Parameters

- **zeff** (*ndarray of float*) – (nat,) electron counts for neutral atoms, generally *Z* nuclear charge. 0 indicates ghosts such that a full fragment of 0s will be constrained to 0 1 charge & multiplicity.
- **fragment_separators** (*ndarray of int*) – (nfr - 1,) indices splitting *zeff* into nfr fragments.
- **molecular_charge** (*float or None*) – Total charge for molecular system.
- **fragment_charges** (*list of float or None*) – (nfr,) known fragment charges with *None* as placeholder for unknown. Expected pre-defaulted so even if nothing known, if *fragment_separators* breaks *zeff* into *nfr=2* fragments, input value should be `fragment_charges=[None, None]`.
- **molecular_multiplicity** (*int or None*) – Total multiplicity for molecular system.
- **fragment_multiplicity** (*list of int or None*) – (nfr,) known fragment charges with *None* as placeholder for unknown. Expected pre-defaulted so even if nothing known, if *fragment_separators* breaks *zeff* into *nfr=2* fragments, input value should be `fragment_multiplicities=[None, None]`.
- **zero_ghost_fragments** (*bool, optional*) – Fragments composed entirely of ghost atoms (*Zeff=0*) are required to have *chgmult 0 1*. When *False*, violations of this will cause a `ValidationError`. When *True*, treat ghost fragments indicated by *zeff* to contain superior information over *chgmult* arguments that might still correspond to full-real molecule. Clears information from *molecular_charge* and *molecular_multiplicity* and sets ghost fragments to 0 1, leaving other positions free to readjust. Unused (prefer to set up such manipulations outside function call) but works.
- **verbose** (*int, optional*) – Amount of printing.

Returns

- **molecular_charge** (*float*) – Total charge for molecular system.
- **fragment_charges** (*list of float*) – (nfr,) Charge on each fragment.
- **molecular_multiplicity** (*int*) – Total multiplicity for molecular system.
- **fragment_multiplicities** (*list of int*) – (nfr,) Multiplicity for each fragment.

Raises `qcelemental.ValidationError` – When no solution to input arguments subject to the constraints below can be found.

Notes

Returns combination of total & fragment charge & multiplicity among values of S1-7 that fulfill rules R1-9. A few derived implications in I1-3.

- Constraints
 - R1 require all chg & mult exist
 - R2 require total charge to be the sum of frag chg
 - R3 require mult is positive int
 - R4 require sufficient tot electrons for mult: $\text{mult} - 1 \leq \text{neut_el} - \text{chg}$
 - R5 require total parity consistent among tot electrons and mult: $(\text{mult} \% 2) \neq ((\text{neut_el} - \text{chg}) \% 2)$
 - R6 require chg match input argument values
 - R7 require mult match input argument values
 - R8 require that $\text{tot} = \text{sum}(\text{frag})$ mult follow high spin addition unless tot & frag mult fully specified
 - R9 require that ghost fragments (zeff all 0) be neutral singlet
- Allowed values
 - S1 suggest input argument values for tot chg, frag chg, tot mult or frag mult
 - S2 suggest sum frag chg for tot chg, allowing for indiv frag chg defaulting to 0
 - S3 suggest distributing unallocated chg onto frag chg
 - S4 suggest 0 default for frag chg
 - S5 suggest range of high-spin sum frag mult for tot mult, allowing for indiv frag mult defaulting to 1 or 2
 - S6 suggest range of unallocated mult = $\text{tot} - \text{high_spin_sum}(\text{frag} - 1)$, allowing for all indiv but self defaulting to 1 or 2.
 - S7 suggest 1 or 2 default for frag mult
- Implications
 - I1 won't form an ion just to be closed shell (would require choosing +1 vs. -1)
 - I2 unallocated chg or mult lands on the first unspecified fragment able to bear it (enforced by returning first match encountered; subsequent matches distribute charge to later frags)
 - I3 missing chg or mult from tot - frags will always be allocated as a block, not distributed

Examples

```

>>> validate_and_fill_chgmult(*sys('He'), 0, [0], 1, [1])
0, [0], 1, [1]
>>> validate_and_fill_chgmult(*sys('He'), None, [None], None, [None])
0, [0], 1, [1]
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, None], None, [None,
↪None])
0, [0, 0], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('He/He'), 2, [None, None], None, [None, None])
2, [2, 0], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('He/He'), None, [2, None], None, [None, None])
2, [2, 0], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('He/He'), 0, [2, None], None, [None, None])
0, [2, -2], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('Ne/He/He'), -2, [None, 2, None], None, [None,
↪None, None])
-2, [-4, 2, 0], 1, [1, 1, 1])
>>> validate_and_fill_chgmult(*sys('Ne/He/He'), 2, [None, -2, None], None, [None,
↪None, None])
2, [4, -2, 0], 1, [1, 1, 1])
# 9 - residual +4 distributes to first fragment able to wholly accept it (He+4
↪is no-go)
>>> validate_and_fill_chgmult(*sys('He/He/Ne'), 2, [None, -2, None], None, [None,
↪None, None])
2, [0, -2, 4], 1, [1, 1, 1])
# 10 - residual +4 unsuited for only open fragment, He, so irreconcilable
>>> validate_and_fill_chgmult(*sys('He/He/Ne'), 2, [None, -2, 0], None, [None,
↪None, None])
ValidationError
# 11 - non-positive multiplicity
>>> validate_and_fill_chgmult(*sys('He/He/Ne'), 2, [2, -2, None], None, [None,
↪None, None])
2, [2, -2, 2], 1, [1, 1, 1])
>>> validate_and_fill_chgmult(*sys('He/He'), None, [-2, 2], None, [None, None])
0, [-2, 2], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, -2], None, [None, None])
-2, [0, -2], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('Ne/Ne'), 0, [None, 4], None, [None, None])
0, [-4, 4], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('He/He/He'), 4, [2, None, None], None, [None,
↪None, None])
4, [2, 2, 0], 1, [1, 1, 1])
>>> validate_and_fill_chgmult(*sys('He/He'), 0, [-2, 2], None, [None, None])
0, [-2, 2], 1, [1, 1])
>>> validate_and_fill_chgmult(*sys('He/He'), 0, [-2, -2], None, [None, None])
ValidationError
>>> validate_and_fill_chgmult(*sys('He'), None, [None], 0, [None])
ValidationError
>>> validate_and_fill_chgmult(*sys('He'), None, [None], None, [1])
0, [0], 1, [1])
# 20 - doublet non consistent with closed-shell, neutral default charge
>>> validate_and_fill_chgmult(*sys('He'), None, [None], None, [2])
ValidationError
>>> validate_and_fill_chgmult(*sys('He'), None, [None], None, [3])
0, [0], 3, [3])
# 22 - insufficient electrons for pentuplet

```

(continues on next page)

(continued from previous page)

```

>>> validate_and_fill_chgmult(*sys('He'), None, [None], None, [5])
ValidationError
>>> validate_and_fill_chgmult(*sys('He'), None, [-1], None, [2])
-1, [-1], 2, [2]
# 24 - doublet not consistent with even charge
>>> validate_and_fill_chgmult(*sys('He'), None, [-2], None, [2])
ValidationError
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, None], None, [1, 1])
0, [0, 0], 1, [1, 1]
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, None], None, [3, 1])
0, [0, 0], 3, [3, 1]
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, None], None, [1, 3])
0, [0, 0], 3, [1, 3]
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, None], None, [3, 3])
0, [0, 0], 5, [3, 3]
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, None], 3, [3, 3])
0, [0, 0], 3, [3, 3]
# 30 - bad parity btwn mult and total # electrons
>>> validate_and_fill_chgmult(*sys('He/He'), None, [None, None], 2, [3, 3])
ValidationError
>>> validate_and_fill_chgmult(*sys('H'), None, [None], None, [None])
0, [0], 2, [2]
>>> validate_and_fill_chgmult(*sys('H'), 1, [None], None, [None])
1, [1], 1, [1]
>>> validate_and_fill_chgmult(*sys('H'), None, [-1], None, [None])
-1, [-1], 1, [1]
>>> validate_and_fill_chgmult(*sys('funnyH'), None, [None], None, [None])
0, [0], 1, [1]
# 35 - insufficient electrons
>>> validate_and_fill_chgmult(*sys('funnierH'), None, [None], None, [None])
ValidationError
>>> validate_and_fill_chgmult(*sys('H/H'), None, [None, None], None, [None, None])
0, [0, 0], 3, [2, 2]
>>> validate_and_fill_chgmult(*sys('H/He'), None, [None, None], None, [None,
↳None])
0, [0, 0], 2, [2, 1]
>>> validate_and_fill_chgmult(*sys('H/He'), None, [1, 1], None, [None, None])
2, [1, 1], 2, [1, 2]
>>> validate_and_fill_chgmult(*sys('H/He'), -2, [-1, None], None, [None, None])
-2, [-1, -1], 2, [1, 2]
>>> validate_and_fill_chgmult(*sys('H/He/Na/Ne'), None, [1, None, 1, None], None,
↳[None, None, None, None])
2, [1, 0, 1, 0], 1, [1, 1, 1, 1]
>>> validate_and_fill_chgmult(*sys('H/He/Na/Ne'), None, [-1, None, 1, None], None,
↳[None, None, None, None])
0, [-1, 0, 1, 0], 1, [1, 1, 1, 1]
>>> validate_and_fill_chgmult(*sys('H/He/Na/Ne'), 2, [None, None, 1, None], None,
↳[None, None, None, None])
2, [1, 0, 1, 0], 1, [1, 1, 1, 1]
>>> validate_and_fill_chgmult(*sys('H/He/Na/Ne'), 3, [None, None, 1, None], None,
↳[None, None, None, None])
3, [0, 2, 1, 0], 2, [2, 1, 1, 1]
>>> validate_and_fill_chgmult(*sys('H/He'), None, [1, None], None, [2, None])
ValidationError
>>> validate_and_fill_chgmult(*sys('H/He'), None, [None, 0], None, [None, 2])
ValidationError
>>> validate_and_fill_chgmult(*sys('H/He'), None, [None, -1], None, [None, 3])

```

(continues on next page)

(continued from previous page)

```

ValidationError
>>> validate_and_fill_chgmult(*sys('H/He/Na/Ne'), None, [None, 1, 0, 1], None,
↳[None, None, None, None])
2, [0, 1, 0, 1], 5, [2, 2, 2, 2]
>>> validate_and_fill_chgmult(*sys('H/He/Na/Ne'), None, [None, 1, 0, None], None,
↳[None, None, None, None])
1, [0, 1, 0, 0], 4, [2, 2, 2, 1]
>>> validate_and_fill_chgmult(*sys('H/He/Na/Ne'), None, [None, 1, 0, None], None,
↳[None, None, 4, None])
1, [0, 1, 0, 0], 6, [2, 2, 4, 1]
>>> validate_and_fill_chgmult(*sys('He/He/He'), 0, [None, None, 1], None, [1,
↳None, 2])
0, [0, -1, 1], 3, [1, 2, 2]
>>> validate_and_fill_chgmult(*sys('N/N/N'), None, [1, 1, 1], 3, [None, 3, None])
3, [1, 1, 1], 3, [1, 3, 1]
>>> validate_and_fill_chgmult(*sys('N/N/N'), None, [1, 1, 1], 3, [None, None,
↳None])
3, [1, 1, 1], 3, [3, 1, 1]
>>> validate_and_fill_chgmult(*sys('N/N/N'), None, [None, None, None], 3, [None,
↳None, 2])
ValidationError
>>> validate_and_fill_chgmult(*sys('N/N/N'), 1, [None, -1, None], 3, [None, None,
↳2])
1, [2, -1, 0], 3, [2, 1, 2]
# 55 - both (1, (1, 0.0, 0.0), 4, (1, 3, 2)) and (1, (0.0, 0.0, 1), 4, (2, 3, 1))
↳plausible
>>> validate_and_fill_chgmult(*sys('N/Ne/N'), 1, [None, None, None], 4, [None, 3,
↳None])
1, [1, 0, 0], 4, [1, 3, 2]
>>> validate_and_fill_chgmult(*sys('N/Ne/N'), None, [None, None, 1], 4, [None, 3,
↳None])
1, [0, 0, 1], 4, [2, 3, 1]
>>> validate_and_fill_chgmult(*sys('He/He'), None, [-1, 1], None, [None, None])
0, [-1, 1], 3, [2, 2]
>>> validate_and_fill_chgmult(*sys('Gh'), 1, [None], None, [None])
ValidationError
>>> validate_and_fill_chgmult(*sys('Gh'), -1, [None], None, [None])
ValidationError
>>> validate_and_fill_chgmult(*sys('Gh'), None, [None], 3, [None])
ValidationError
>>> validate_and_fill_chgmult(*sys('He/Gh'), None, [2, None], None, [None, None])
2, [2, 0], 1, [1, 1]
>>> validate_and_fill_chgmult(*sys('Gh/He'), None, [2, None], None, [None, None])
ValidationError
>>> validate_and_fill_chgmult(*sys('Gh/He/Ne'), 2, [None, -2, None], None, [None,
↳None, None])
2, [0, -2, 4], 1, [1, 1, 1]
>>> validate_and_fill_chgmult(*sys('Gh/He/Gh'), 1, [None, None, None], None,
↳[None, None, None])
1, [0, 1, 0], 2, [1, 2, 1]
>>> sys = {
    'He': (np.array([2]), np.array([])),
    'He/He': (np.array([2, 2]), np.array([1])),
    'Ne/He/He': (np.array([10, 2, 2]), np.array([1, 2])),
    'He/He/Ne': (np.array([2, 2, 10]), np.array([1, 2])),
    'Ne/Ne': (np.array([10, 10]), np.array([1])),
    'He/He/He': (np.array([2, 2, 2]), np.array([1, 2])),

```

(continues on next page)

(continued from previous page)

```
'H': (np.array([1]), np.array([])),
'funnyH': (np.array([0]), np.array([])), # has no electrons
'funnierH': (np.array([-1]), np.array([])), # has positron
'H/H': (np.array([1, 1]), np.array([1])),
'H/He': (np.array([1, 2]), np.array([1])),
'H/He/Na/Ne': (np.array([1, 2, 11, 10]), np.array([1, 2, 3])),
'N/N/N': (np.array([7, 7, 7]), np.array([1, 2])),
'N/Ne/N': (np.array([7, 10, 7]), np.array([1, 2])),
'He/Gh': (np.array([2, 0]), np.array([1])),
'Gh/He': (np.array([0, 2]), np.array([1])),
'Gh': (np.array([0, 0]), np.array([])),
'Gh/He/Ne': (np.array([0, 0, 2, 10]), np.array([2, 3])),
'Gh/He/Gh': (np.array([0, 2, 0]), np.array([1, 2]))}
```

4.7.5 qcelemental.testing Module

Functions

<code>compare(expected, computed[, label, quiet, ...])</code>	Returns True if two integers, strings, booleans, or integer arrays are element-wise equal.
<code>compare_molrecs(expected, computed[, label, ...])</code>	Function to compare Molecule dictionaries.
<code>compare_recursive(expected, computed[, ...])</code>	Recursively compares nested structures such as dictionaries and lists.
<code>compare_values(expected, computed[, label, ...])</code>	Returns True if two floats or float arrays are element-wise equal within a tolerance.
<code>tnm()</code>	Returns the name of the calling function, usually name of test case.

compare

`qcelemental.testing.compare` (*expected, computed, label: str = None, *, quiet: bool = False, return_message: bool = False, return_handler: Callable = None*) → bool

Returns True if two integers, strings, booleans, or integer arrays are element-wise equal.

Parameters

- **expected** (*int, bool, str or int array-like*) – Reference value against which *computed* is compared.
- **computed** (*int, bool, str or int array-like*) – Input value to compare against *expected*.
- **label** (*str, optional*) – Label for passed and error messages. Defaults to calling function name.

Returns

- **allclose** (*bool*) – Returns True if *expected* and *computed* are equal; False otherwise.
- **message** (*str, optional*) – When *return_message=True*, also return passed or error message.

Other Parameters **return_handler** (*function, optional*) – Function to control printing, logging, raising, and returning. Specialized interception for interfacing testing systems.

Notes

- Akin to `np.array_equal`.
- For scalar exactly-comparable types and for arbitrary-dimension, `np.ndarray-castable`, `uniform-type`, exactly-comparable types. For mixed types, use `compare_recursive()`.

compare_molrecs

`qcelemental.testing.compare_molrecs` (*expected*, *computed*, *label*: *str* = *None*, *, *atol*: *float* = *1e-06*, *rtol*: *float* = *1e-16*, *forgive*=*None*, *verbose*: *int* = *1*, *relative_geoms*='exact', *return_message*: *bool* = *False*, *return_handler*: *Callable* = *None*) → *bool*

Function to compare Molecule dictionaries. Prints `#util.success()` when elements of *computed* match elements of *expected* to *tol* number of digits (for float arrays).

compare_recursive

`qcelemental.testing.compare_recursive` (*expected*, *computed*, *label*: *str* = *None*, *, *atol*: *float* = *1e-06*, *rtol*: *float* = *1e-16*, *forgive*: *bool* = *None*, *quiet*: *bool* = *False*, *return_message*: *bool* = *False*, *return_handler*: *Callable* = *None*) → *bool*

Recursively compares nested structures such as dictionaries and lists.

Parameters

- **expected** (*dict*) – Reference value against which *computed* is compared. Dict may be of any depth but should contain Plain Old Data.
- **computed** (*int*, *bool*, *str* or *int array-like*) – Input value to compare against *expected*. Dict may be of any depth but should contain Plain Old Data.
- **atol** (*int* or *float*, *optional*) – Absolute tolerance (see formula below).
- **label** (*str*, *optional*) – Label for passed and error messages. Defaults to calling function name.
- **rtol** (*float*, *optional*) – Relative tolerance (see formula below). By default set to zero so *atol* dominates.
- **forgive** (*list*, *optional*) – Keys in top level which may change between *expected* and *computed* without triggering failure.

Returns

- **allclose** (*bool*) – Returns True if *expected* and *computed* are equal within tolerance; False otherwise.
- **message** (*str*, *optional*) – When *return_message*=True, also return passed or error message.

Notes

$$\text{absolute}(\text{computed} - \text{expected}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{expected}))$$

compare_values

`qcelestial.testing.compare_values` (*expected, computed, label: str = None, *, atol: float = 1e-06, rtol: float = 1e-16, equal_nan: bool = False, passnone: bool = False, quiet: bool = False, return_message: bool = False, return_handler: Callable = None*) → bool

Returns True if two floats or float arrays are element-wise equal within a tolerance.

Parameters

- **expected** (*float or float array-like*) – Reference value against which *computed* is compared.
- **computed** (*float or float array-like*) – Input value to compare against *expected*.
- **atol** (*float, optional*) – Absolute tolerance (see formula below).
- **label** (*str, optional*) – Label for passed and error messages. Defaults to calling function name.
- **rtol** (*float, optional*) – Relative tolerance (see formula below). By default set to zero so *atol* dominates.
- **equal_nan** (*bool, optional*) – Passed to `np.isclose`. Compare NaN's as equal.
- **passnone** (*bool, optional*) – Return True when both *expected* and *computed* are None.
- **quiet** (*bool, optional*) – Whether to log the return message.
- **return_message** (*bool, optional*) – Whether to return tuple. See below.

Returns

- **allclose** (*bool*) – Returns True if *expected* and *computed* are equal within tolerance; False otherwise.
- **message** (*str, optional*) – When `return_message=True`, also return passed or error message.

Other Parameters **return_handler** (*function, optional*) – Function to control printing, logging, raising, and returning. Specialized interception for interfacing testing systems.

Notes

- Akin to `np.allclose`.
- For scalar float-comparable types and for arbitrary-dimension, `np.ndarray-castable`, uniform-type, float-comparable types. For mixed types, use `compare_recursive()`.
- Sets `rtol` to zero to match expected Psi4 behaviour, otherwise measured as:

$$\text{absolute}(\text{computed} - \text{expected}) \leq (\text{atol} + \text{rtol} * \text{absolute}(\text{expected}))$$

tnm

`qcelestial.testing.tnm()` → str

Returns the name of the calling function, usually name of test case.

4.8 Changelog

4.8.1 0.6.1 / 2019-08-19

Bug Fixes

- (GH#114) The Numpy einsum calls reference the top level functions and not core C functions. This fixes an issue which can result in NumPy version dependencies.

4.8.2 0.6.0 / 2019-08-14

New Features

- (GH#85, GH#87) Msgpack is a new serialization option for Models. Serialization defaults to msgpack when available (`conda install msgpack-python [-c conda-forge]`), falling back to JSON otherwise. This results in substantial speedups for both serialization and deserialization actions and should be a transparent replacement for users within Elemental itself.

Enhancements

- (GH#78) Molecular alignments can now be aligned on the derivatives of vector components.
- (GH#81) Testing is now operated both on the minimal supported and the latest released versions of dependencies.
- (GH#82) Molecule fragment grouping is now disabled by default to match expected behavior.
- (GH#84) Testing without internet connection should now pass since PubChem testing is skipped with no connection.
- (GH#85) Molecule switches from lists to numpy arrays for internal storage of per-atom fields.
- (GH#86) Molecule performance and memory enhancements through reduced validation times and LRU caching of common validations.
- (GH#88, GH#109) The Molecule Model now has its attributes documented and in an on-the-fly manner derived from the Pydantic Schema of those attributes.
- (GH#99, GH#100, GH#101, GH#102, GH#103, GH#104, GH#105, GH#106, GH#107) Various documentation, type hints, and small changes.

Bug Fixes

- (GH#87) Molecule objects built from Schema are run through validators for consistency.

4.8.3 0.5.0 / 2019-07-16

Enhancements

- (GH#76) Adds a built-in `Molecule.to_file` function for easy serialization into `.numpy`, `.json`, `.xyz`, `.psimol`, and `.psi4` file formats.

Bug Fixes

- (GH#74) Atom and fragment ordering are preserved when invoking `get_fragment`.

4.8.4 0.4.2 / 2019-06-13

New Features

- (GH#70, GH#72) `molparse.to_string` Molpro dtype developed.

4.8.5 0.4.1 / 2019-05-31

New Features

Enhancements

- (GH#68) `molparse.to_string` learned parameter `return_data` that contains aspects of the `models.Molecule` not expressible in the string. Implemented for dtypes `xyz`, `cfour`, `psi4`.
- (GH#68) `Datum` gained an attribute `numeric` that reflects whether arithmetic on data is valid. `Datums` that aren't numeric can now be created by initializing with `numeric=False`.

Bug Fixes

- (GH#66) Fix tests when `networkx` not installed.
- (GH#67) Fix “unsupported format string passed to `numpy.ndarray.__format__`” on Mac for testing. `compare_values`.

4.8.6 0.4.0 / 2019-05-13

New Features

- (GH#51) Changes `models.Molecule` connectivity to default to `None` rather than an empty list. **WARNING** this change alters the hashes produced from the `Molecule.get_hash` functionality.
- (GH#52, GH#53) `models.Molecule` learned `nuclear_repulsion_energy`, `nelectrons`, and `to_string` functions.
- (GH#54) `models.ResultProperties` supports CCSD and CCSD(T) properties.
- (GH#56) Algorithms `Kabsch` `molutil.kabsch_align`, Hungarian `util.linear_sum_assignment`, and `Uno` `util.uno` added. Utilities to generate random 3D rotations `util.random_rotation_matrix` and `reindex` a NumPy array into smaller blocks `util.blockwise_expand` added.
- (GH#56) Molecular alignment taking into account displacement, rotation, atom exchange, and mirror symmetry for superimposable and differing geometries was added in `molutil.B787` (basis NumPy function) and `models.Molecule.align` (far more convenient). Suitable for QM-sized molecules. Requires addition package `networkx`.
- (GH#58) `utils` learned `which_import` and `which` that provide a path or boolean result for locating modules or commands, respectively. These were migrated from `QCEngine` along with `safe_version` and `parse_version` to colocate the import utilities.

- (GH#61) Add molecular visualization to the `models.Molecule` object through the optional 3dMol.js framework.
- (GH#65) `testing.compare_molrecs` learned parameter `relative_geoms='align'` that lets Molecules pass if geometries within a translation and rotation of each other.
- (GH#65) `testing.compare_recursive` learned parameter `forgive` that is a list of paths that may differ without failing the comparison.

Enhancements

- (GH#52, GH#53) `molparse.to_string` NWChem and GAMESS dtypes developed.
- (GH#57) `molparse.to_string` learned `dtype='terachem'` for writing the separate XYZ file required by TeraChem. Angstroms or Bohr allowed, though the latter requires extra in input file.
- (GH#60) `util.which` added the Python interpreter path to the default search `$PATH`.
- (GH#62) Added `*` to parameter list of many functions requiring subsequent to be keyword only. Code relying heavily on positional arguments may get broken.
- (GH#63) `util.which` learned parameter `env` to use an alternate search `$PATH`.
- (GH#63) `util.which` and `util.which_import` learned parameters `raise_error` and `raise_msg` which raises `ModuleNotFoundError` (for both functions) when not located. It error will have a generic error message which can be extended by `raise_msg`. It is strongly encouraged to add specific remedies (like how to install) through this parameter. This is the third exit pattern possible from the “which” functions, of which `path/None` is the default, `True/error` happens when `raise_error=True`, and `True/False` happens otherwise when `return_bool=True`.
- (GH#65) Testing functions `compare`, `compare_values`, `compare_recursive` learned parameter `return_handler` that lets other printing, logging, and pass/fail behavior to be interjected.

Bug Fixes

- (GH#63) `util.which` uses `os.pathsep` rather than Linux-focused `..`.
- (GH#65) Fixed some minor printing and tolerance errors in molecule alignment.
- (GH#65) `testing.compare_recursive` stopped doing `atol=10**-atol` for `atol>=1`, bringing it in line with other compare functions.

4.8.7 0.3.3 / 2019-03-12

Enhancements

- (GH#49) Precompute some mass number and mass lookups and store on `qcel.periodic_table`. Also move static `re.compile` expressions out of fns on to module. Mol validation `.127s` → `.005s`.

4.8.8 0.3.2 / 2019-03-11

New Features

- (GH#47) `models.DriverEnum` now has a `derivative_int` function to return 1 for gradient, etc., for easy math. `properties` returns 0.

- (GH#47) Optional `fix_symmetry` field in `qcschema_molecule` was missing from `models.Molecule` so Pydantic got mad at Psi4. Now calmed.

Enhancements

- (GH#48) If `Molecule` object has passed through `molparse` validation because it was created with a `molparse` constructor (e.g., `from_string`), save some time by not passing it through again at `model.Molecule` creation time.

Bug Fixes

- (GH#48) Fixed a `Molecule.get_fragment` bug where ghosted fragments still asserted charge/multiplicity to the validator, which was rightly confused.

4.8.9 0.3.1 / 2019-03-07

Enhancements

- (GH#37) Documentation now pulls from the custom QC Archive Sphinx Theme, but can fall back to the standard RTD theme. This allows all docs across QCA to appear consistent with each other.
- (GH#41) Conda-build recipe removed to avoid possible confusion for everyone who isn't a Conda-Forge recipe maintainer. Tests now rely on the `conda env` setups.
- (GH#44) `Molecule` objects are now always validated against a more rigorous model and fragment multiplicities are fixed at the correct times, even when no multiplicities are provided. `Molecule` defaults to `dtype=2`.

Bug Fixes

- (GH#39) Fixed `setup.py` to call `pytest` instead of `unittest` when running tests on install
- (GH#41) Pinned a minimum `Pytest` version to make sure errors are not because of too old of a `pytest` version

4.8.10 0.3.0 / 2019-02-27

New Features

- (GH#33) `molparse.to_schema` recognizes `dtype=2` in keeping with `GH:MoISSI/QCSchema#60` with internal `schema_name=qcschema_molecule` and `schema_version=2` fields. `molparse.from_schema` recognizes external fields (existing functionality), internal fields (`dtype=2`), and mixed.
- (GH#33) Pydantic molecule model now contains `schema_name` and `schema_version=2` information.
- (GH#35) Models now have an `extra` field for extra attributes, no additional base keys are allowed.

Enhancements

- (GH#34) Converts `qcel.Datum` to Pydantic model. Changes: (a) `comment`, `doi`, `glossary` fields must be accessed by keyword, (b) `to_dict()` becomes `dict()` and instead of only `label`, `units`, `data` fields in `dict`, now `comment`, `doi`, `glossary` `present_if_non-default`, (c) complex values no longer list-ified by `to_dict()`.
- (GH#36) Changelog and Models documentation.

Bug Fixes

4.8.11 0.2.6 / 2019-02-18

Bug Fixes

- (GH#32) Updates compliance with Pydantic v0.20.

4.8.12 0.2.5 / 2019-02-13

Enhancements

- (GH#31) Lints the code base preparing for a release and minor test improvements.

Bug Fixes

- (GH#30) Fixes `dihedral` measurement code for incorrect phase in certain quadrants.

4.8.13 0.2.4 / 2019-02-08

New Features

- (GH#27) Adds a new `measure` feature to `Molecule` for distances, angles, and dihedrals.
- (GH#25) Adds a new `testing` module which contains testing routines for arrays, dictionaries, and molecules.

Enhancements

- (GH#28) Reduces loading time from ~1 second to 200 ms by delaying `pint` import and ensuring git tags are only computed once.

4.8.14 0.2.3 / 2019-01-29

Enhancements

- (GH#24) Update models to be compatible with `QCFractal` and `MongoDB` objects in the `QCArchive Ecosystem`. Also enhances the `Molecule` model's `json` function to accept `as_dict` keyword, permitting a return as a dictionary of Pydantic-serialized python (primitive) objects, instead of a string.

4.8.15 0.2.2 / 2019-01-28

Bug Fixes

- (GH#21) `Molparse`'s `from_schema` method now correctly parses the new `qcschema_X` strings for schema names.
- (GH#23) Pydantic model serializations now correctly handle `Numpy Array` objects in nested `BaseModels`. Model serialization testing added to catch these in the future.

4.8.16 0.2.1 / 2019-01-27

- (GH#20) Moves several Molecule parsing functions to the molparse module.

4.8.17 0.2.0 / 2019-01-25

- now requires Python 3.6+
- now requires Pydantic

New Features

- (GH#14, GH#16, GH#17) Added new Pydantic models for Molecules, Results, and Optimizations to make common objects used in the QCArchive project all exist in one central, always imported module.

Enhancements

- (GH#13) Function `util.unnp` that recursively list-ifies ndarray in a dict now handles lists and flattens.

4.8.18 0.1.3 / 2018-12-14

New Features

- (GH#12) Adds “connectivity” validation and storage consistent with QCSchema.

Enhancements

- (GH#12) Adds single dictionary provenance consistent with QCSchema rather than previous list o’dicts.

4.8.19 0.1.2 / 2018-11-3

New Features

- (GH#10) Adds covalent radii data available through `covalentradii.get(atom)` function.
- (GH#10) Adds `to_units(unit)` to Datum class to access the data in non-native units.
- (GH#10) Adds `periodictable.to_period(atom)` and `to_group(atom)` functions to address periodic table.

4.8.20 0.1.1 / 2018-10-30

New Features

- (GH#7, GH#9) Adds “comment” and “provenance” fields to internal repr to better match QCSchema.
- (GH#7) Adds provenance stamp to `from_string`, `from_arrays`, `from_schema` functions.

Enhancements

- (GH#7) Adds outer `schema_name/schema_version` to `to_schema(..., dtype=1)` output so is inverse to `from_schema`.

Bug Fixes

- (GH#8) Tests pass for installed module now that comparison tests are xfail.

4.8.21 0.1.0a / 2018-10-24

This is the first alpha release of QCElemental containing the primary three components.

New Features

- (GH#6) Updated `molparse` to write new Molecule QCSchema fields in keeping with GH:MoISSI/QCSchema#44
- Periodic Table data from NIST SRD144 (c. pre-2015?) collected into `qcelemental.periodictable` instance, with accessors `to_Z`, `to_element`, `to_E`, `to_mass`, `to_A` (and redundant accessors `to_mass_number`, `to_atomic_number`, `to_symbol`, `to_name`) in `float` and `Decimal` formats. Also includes functionality to write a corresponding “C” header.
- Physical Constants data from NIST SRD121 (CODATA 2014) collected into `qcelemental.constants` instance, with access through `qcelemental.constants.Faraday_constant` (exact capitalization; `float` result) or `get` (free capitalization; `float` or `Decimal` result). Also includes functionality to write a corresponding “C” header.
- `molparse` submodule where `from_string`, `from_array`, `from_schema` constructors parse and rearrange (if necessary) and validate molecule topology inputs from the QC and EFP domains into a QCSchema-like data structure. Current deficiencies from QCSchema are non-contiguous fragments and “provenance” fields. Accessors `to_string` and `to_schema` are highly customizable.
- A `pint` context has been built around the NIST physical constants data so that `qcelemental.constants.conversion_factor(from_unit, to_unit)` uses the QCElemental values in its conversions. Resulting `float` is within uncertainty range of NIST constants but won’t be exact for conversions involving multiple fundamental dimensions or `wavelength -> energy != 1 / (energy -> wavelength)`.

4.9 Indices and tables

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

q

`qcelemental`, [22](#)

`qcelemental.molparse`, [38](#)

`qcelemental.periodic_table`, [32](#)

`qcelemental.physical_constants`, [35](#)

`qcelemental.testing`, [54](#)

A

A (in module *qcelemental*), 30
 A (*qcelemental.periodic_table.PeriodicTable* attribute), 32

C

ChoicesError, 25
 comment (*qcelemental.Datum* attribute), 27
 compare() (in module *qcelemental*), 22
 compare() (in module *qcelemental.testing*), 54
 compare_molrecs() (in module *qcelemental.testing*), 55
 compare_recursive() (in module *qcelemental*), 23
 compare_recursive() (in module *qcelemental.testing*), 55
 compare_values() (in module *qcelemental*), 23
 compare_values() (in module *qcelemental.testing*), 56
 constants (in module *qcelemental*), 30
 contiguize_from_fragment_pattern() (in module *qcelemental.molparse*), 38
 conversion_factor (class in *qcelemental.constants*), 13
 conversion_factor (*qcelemental.physical_constants.PhysicalConstantsContext* attribute), 37
 conversion_factor (*qcelemental.PhysicalConstantsContext* attribute), 28
 CovalentRadii (class in *qcelemental*), 25
 cr (*qcelemental.CovalentRadii* attribute), 25

D

data (*qcelemental.Datum* attribute), 27
 DataUnavailableError, 26
 Datum (class in *qcelemental*), 27
 doi (in module *qcelemental*), 30
 doi (*qcelemental.CovalentRadii* attribute), 25
 doi (*qcelemental.Datum* attribute), 27
 doi (*qcelemental.physical_constants.PhysicalConstantsContext* attribute), 36

doi (*qcelemental.PhysicalConstantsContext* attribute), 28

E

E (in module *qcelemental*), 30
 E (*qcelemental.periodic_table.PeriodicTable* attribute), 32
 EA (in module *qcelemental*), 31
 EA (*qcelemental.periodic_table.PeriodicTable* attribute), 32

F

from_arrays() (in module *qcelemental.molparse*), 39
 from_input_arrays() (in module *qcelemental.molparse*), 41
 from_schema() (in module *qcelemental.molparse*), 42
 from_string() (in module *qcelemental.molparse*), 42

G

get() (in module *qcelemental.constants*), 13
 get() (*qcelemental.CovalentRadii* method), 25
 get() (*qcelemental.physical_constants.PhysicalConstantsContext* method), 37
 get() (*qcelemental.PhysicalConstantsContext* method), 29
 glossary (*qcelemental.Datum* attribute), 27

L

label (*qcelemental.Datum* attribute), 27

M

mass (in module *qcelemental*), 31
 mass (*qcelemental.periodic_table.PeriodicTable* attribute), 32
 Molecule (class in *qcelemental.models*), 20
 MoleculeFormatError, 27

N

name (in module *qcelemental*), 30, 31

- name (*qcelemental.CovalentRadii* attribute), 25
- name (*qcelemental.periodic_table.PeriodicTable* attribute), 32
- name (*qcelemental.physical_constants.PhysicalConstantsContext* attribute), 36
- name (*qcelemental.PhysicalConstantsContext* attribute), 28
- native_units (*qcelemental.CovalentRadii* attribute), 25
- NotAnElementError, 27
- numeric (*qcelemental.Datum* attribute), 27
- ## P
- parse_nucleus_label() (in module *qcelemental.molparse*), 44
- pc (in module *qcelemental*), 30
- pc (*qcelemental.physical_constants.PhysicalConstantsContext* attribute), 36
- pc (*qcelemental.PhysicalConstantsContext* attribute), 28
- PeriodicTable (class in *qcelemental.periodic_table*), 32
- periodictable (in module *qcelemental*), 30
- PhysicalConstantsContext (class in *qcelemental*), 27
- PhysicalConstantsContext (class in *qcelemental.physical_constants*), 36
- ## Q
- qcelemental (module), 22
- qcelemental.molparse (module), 38
- qcelemental.periodic_table (module), 32
- qcelemental.physical_constants (module), 35
- qcelemental.testing (module), 54
- Quantity() (in module *qcelemental.constants*), 13
- Quantity() (*qcelemental.physical_constants.PhysicalConstantsContext* method), 37
- Quantity() (*qcelemental.PhysicalConstantsContext* method), 29
- ## R
- raw_codata (in module *qcelemental*), 30
- raw_codata (*qcelemental.physical_constants.PhysicalConstantsContext* attribute), 36
- raw_codata (*qcelemental.PhysicalConstantsContext* attribute), 28
- reconcile_nucleus (*qcelemental.molparse* attribute), 45
- run_comparison() (*qcelemental.periodic_table.PeriodicTable* method), 33
- run_comparison() (*qcelemental.physical_constants.PhysicalConstantsContext* method), 37
- run_comparison() (*qcelemental.PhysicalConstantsContext* method), 29
- ## S
- string_representation() (in module *qcelemental.constants*), 14
- string_representation() (*qcelemental.CovalentRadii* method), 26
- string_representation() (*qcelemental.physical_constants.PhysicalConstantsContext* method), 37
- string_representation() (*qcelemental.PhysicalConstantsContext* method), 29
- ## T
- tnm() (in module *qcelemental.testing*), 56
- to_A() (*qcelemental.periodic_table.PeriodicTable* method), 33
- to_atomic_number() (in module *qcelemental.periodictable*), 15
- to_atomic_number() (*qcelemental.periodic_table.PeriodicTable* method), 33
- to_E() (*qcelemental.periodic_table.PeriodicTable* method), 33
- to_element() (*qcelemental.periodic_table.PeriodicTable* method), 34
- to_group() (in module *qcelemental.periodictable*), 16
- to_group() (*qcelemental.periodic_table.PeriodicTable* method), 34
- to_mass() (in module *qcelemental.periodictable*), 15
- to_mass() (*qcelemental.periodic_table.PeriodicTable* method), 34
- to_mass_number() (in module *qcelemental.periodictable*), 15
- to_mass_number() (*qcelemental.periodic_table.PeriodicTable* method), 34
- to_name() (in module *qcelemental.periodictable*), 16
- to_name() (*qcelemental.periodic_table.PeriodicTable* method), 35
- to_period() (in module *qcelemental.periodictable*), 16
- to_period() (*qcelemental.periodic_table.PeriodicTable* method), 35
- to_schema() (in module *qcelemental.molparse*), 48
- to_string() (in module *qcelemental.molparse*), 48

`to_symbol()` (in module *qcelemental.periodictable*),
15

`to_symbol()` (*qcelemental.periodic_table.PeriodicTable* method),
35

`to_Z()` (*qcelemental.periodic_table.PeriodicTable* method), 33

U

`units` (*qcelemental.Datum* attribute), 27

`ureg` (*qcelemental.physical_constants.PhysicalConstantsContext* attribute), 37

`ureg` (*qcelemental.PhysicalConstantsContext* attribute),
29

V

`validate_and_fill_chgmult()` (in module *qcelemental.molparse*), 49

`ValidationError`, 29

W

`write_c_header()` (*qcelemental.CovalentRadii* method), 26

`write_c_header()` (*qcelemental.periodic_table.PeriodicTable* method),
35

`write_c_header()` (*qcelemental.physical_constants.PhysicalConstantsContext* method), 38

`write_c_header()` (*qcelemental.PhysicalConstantsContext* method), 29

Y

`year` (in module *qcelemental*), 30

`year` (*qcelemental.CovalentRadii* attribute), 25

`year` (*qcelemental.physical_constants.PhysicalConstantsContext* attribute), 36

`year` (*qcelemental.PhysicalConstantsContext* attribute),
28

Z

`Z` (in module *qcelemental*), 30

`Z` (*qcelemental.periodic_table.PeriodicTable* attribute),
32