QCElemental Documentation

Release v0.12.0+0.g3dbdd32.dirty

The QCArchive Development Team

Dec 01, 2019

GETTING STARTED

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

CHAPTER

ONE

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
```

CHAPTER

TWO

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
```

CHAPTER

THREE

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": ...}
```

CHAPTER

FOUR

INDEX

Getting Started

4.1 Install QCElemental

You can install qcelemental with `conda` or with `pip`.

4.1.1 Conda

You can install qcelemental using conda:

```
>>> conda install qcelemental -c conda-forge
```

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

4.1.2 Pip

You can also install QCElemental using `pip`:

```
>>> pip install qcelemental
```

4.1.3 Test the Installation

You can test to make sure that Elemental is installed correctly by first installing `pytest`.

From `conda`:

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

From `pip`:

```
>>> pip install pytest
```

Then, run the following command:

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

4.1.4 Developing from Source

If you are a developer and want to make contributions Elemental, you can access the source code from github.

Quantities

- *Physical Constants*
- *Periodic Table*
- *Covalent Radii*
- *van der Waals Radii*

4.2 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.

4.2.1 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 shorthands.

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

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

4.2.2 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')>
```

4.2.3 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'
```

4.2.4 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.

4.2.5 API

Top level user functions:

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

4.2.6 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']

`qcelemental.constants.`**`Quantity`**(*data: str*) → quantity._Quantity
Returns a Pint Quantity.

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

4.3 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 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 infomation 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
```

4.3.1 API

Top level user functions:

to_mass(atom, *[, return_decimal])	Get atomic mass of *atom*.
to_mass_number(atom)	Get mass number of *atom*.
to_atomic_number(atom)	Get atomic number of *atom*.
to_symbol(atom)	Get element symbol of *atom*.
to_name(atom)	Get element name of *atom*.
to_period(atom)	Get period (horizontal row in periodic table) of *atom*.
to_group(atom)	Get group (vertical column in periodic table) of *atom*.

4.3.2 Function Definitions

qcelemental.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.

qcelemental.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.

qcelemental.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.

qcelemental.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.

4.4 Covalent Radii

Access Covalent Radii information within QCElemental.

```
>>> qcel.covalentradii.get(6)
1.4361918553479494
>>> qcel.covalentradii.get("C")
1.4361918553479494
>>> qcel.covalentradii.get("C12")
1.4361918553479494
>>> qcel.covalentradii.get("Carbon")
1.4361918553479494
```

4.4.1 Contexts

To prepare for future changes, covalent radii are contained in contexts. The `qcel.covalentradii` context will be updated over time to the latest data. To "pin" a context version, a specific context can be created like so:

```
>>> context = qcel.CovalentRadii("ALVAREZ2008")
>>> qcel.covalentradii.get(6)
1.4361918553479494
```

Currently only `ALVAREZ2008` is available.

4.4.2 API

Top level user functions:

get(atom, *[, return_tuple, units, missing])	Access a covalent radius for species *atom*.
string_representation()	Print name, value, and units of all covalent radii.

4.4.3 Function Definitions

qcelemental.covalentradii.**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:
    rad = qcel.covalentradii.get(atom)
except qcel.DataUnavailableError:
    rad = 4.0
```

  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.

- *qcelemental.Datum* – When `return_tuple=True`, Datum 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*.

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

4.5 van der Waals Radii

Access van der Waals Radii information within QCElemental.

```
>>> qcel.vdwradii.get(6)
3.212534413278308
>>> qcel.vdwradii.get("C")
3.212534413278308
>>> qcel.vdwradii.get("C12")
3.212534413278308
>>> qcel.vdwradii.get("Carbon")
3.212534413278308
```

4.5.1 Contexts

To prepare for future changes, van der waals radii are contained in contexts. The `qcel.vdwradii` context will be updated over time to the latest data. To "pin" a context version, a specific context can be created like so:

```
>>> context = qcel.VanderWaalsRadii("MANTINA2009")
>>> qcel.vdwradii.get(6)
3.212534413278308
```

Currently only `MANTINA2009` is available.

4.5.2 API

Top level user functions:

get(atom, *[, return_tuple, units, missing])	Access a van der Waals radius for species `atom`.
string_representation()	Print name, value, and units of all van der Waals radii.

4.5.3 Function Definitions

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

Parameters

- **atom** (*int or str*) – Identifier for element or nuclide, e.g., `H`, `C`, `Al`.
- **units** (*str, optional*) – Units of returned value. To return in native unit (MANTINA2009: 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:
    rad = qcel.vdwradii.get(atom)
except qcel.DataUnavailableError:
    rad = 4.0
```

  Only relevant for `return_tuple=False`.
- **return_tuple** (*bool, optional*) – See below.

Returns

- *float* – When `return_tuple=False`, value of Van der Waals radius. If multiple defined for element, returns largest.
- *qcelemental.Datum* – When `return_tuple=True`, Datum with units, description, uncertainty, and value of van der Waals 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 van der Waals radius is available and *missing=None*.

qcelemental.vdwradii.**string_representation**() → str
Print name, value, and units of all van der Waals radii.

QCSchema Models

Implementation descriptions of QCSchema objects in Python.

- *Overview*
- *Molecule*
- *AtomicResult*
- *Common*

4.6 Overview

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

4.6.1 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

>
```

4.6.2 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]}
```

4.7 Molecule

A Python implementation 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.

4.7.1 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
>
```

4.7.2 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'
```

4.7.3 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 aquired 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)      3.100000000572     0.000000000000     0.000000000000
>
```

4.7.4 Fields

class qcelemental.models.**Molecule**(*orient: bool = False*, *validate: Optional[bool] = None*, ***kwargs: Any*)

A QCSchema representation of a Molecule. This model contains data for symbols, geometry, connectivity, charges, fragmentation, etc while also supporting a wide array of I/O and manipulation capabilities.

Molecule objects geometry, masses, and charges are truncated to 8, 6, and 4 decimal places respectively to assist with duplicate detection.

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*nat, or the serialized version of the array in (3*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 of strings of shape (nat,). Typical use is in model conversions, such as Elemental <-> 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': 'v0.12.0+0.g3dbdd32.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.

4.7.5 API

class qcelemental.models.**Molecule**(*orient: bool = False*, *validate: Optional[bool] = None*, ***kwargs: Any*)

A QCSchema representation of a Molecule. This model contains data for symbols, geometry, connectivity, charges, fragmentation, etc while also supporting a wide array of I/O and manipulation capabilities.

Molecule objects geometry, masses, and charges are truncated to 8, 6, and 4 decimal places respectively to assist with duplicate detection.

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*nat, or the serialized version of the array in (3*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 of strings of shape (nat,). Typical use is in model conversions, such as Elemental <-> 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 preprocesses 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': 'v0.12.0+0.g3dbdd32.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.

align(*ref_mol*, *, *do_plot=False*, *verbose=0*, *atoms_map=False*, *run_resorting=False*, *mols_align=False*, *run_to_completion=False*, *uno_cutoff=0.001*, *run_mirror=False*)

Finds shift, rotation, and atom reordering of *concern_mol* (self) that best aligns with *ref_mol*.

Wraps `qcel.molutil.B787()` for `qcel.models.Molecule`. Employs the Kabsch, Hungarian, and Uno algorithms to exhaustively locate the best alignment for non-oriented, non-ordered structures.

Parameters

- **concern_mol** (*qcel.models.Molecule*) – Molecule of concern, to be shifted, rotated, and reordered into best coincidence with *ref_mol*.
- **ref_mol** (*qcel.models.Molecule*) – Molecule to match.
- **atoms_map** (*bool, optional*) – Whether atom1 of *ref_mol* corresponds to atom1 of *concern_mol*, etc. If true, specifying *True* can save much time.
- **mols_align** (*bool, optional*) – Whether *ref_mol* and *concern_mol* have identical geometries by eye (barring orientation or atom mapping) and expected final RMSD = 0. If *True*, procedure is truncated when RMSD condition met, saving time.
- **do_plot** (*bool, optional*) – Pops up a mpl plot showing before, after, and ref geometries.
- **run_to_completion** (*bool, optional*) – Run reorderings to completion (past RMSD = 0) even if unnecessary because *mols_align=True*. Used to test worst-case timings.
- **run_resorting** (*bool, optional*) – Run the resorting machinery even if unnecessary because *atoms_map=True*.
- **uno_cutoff** (*float, optional*) – TODO
- **run_mirror** (*bool, optional*) – Run alternate geometries potentially allowing best match to *ref_mol* from mirror image of *concern_mol*. Only run if system confirmed to be nonsuperimposable upon mirror reflection.
- **verbose** (*int, optional*) – Print level.

Returns

Molecule is internal geometry of *self* optimally aligned and atom-ordered to *ref_mol*. Presently all fragment information is discarded.

data['rmsd'] is RMSD [A] between *ref_mol* and the optimally aligned geometry computed. *data['mill']* is a AlignmentMill with fields (shift, rotation, atommap, mirror) that prescribe the transformation from *concern_mol* and the optimally aligned geometry.

Return type *Molecule*, data

compare(*other*)

Compares the current object to the provided object recursively.

Parameters

- **other** (*Model*) – The model to compare to.
- ****kwargs** – Additional kwargs to pass to `qcelemental.compare_recursive`.

Returns True if the objects match.

Return type bool

classmethod from_data(*data: Union[str, Dict[str, Any], numpy.array, bytes], dtype: Optional[str] = None, *, orient: bool = False, validate: bool = None, **kwargs: Dict[str, Any]*) → qcelemental.models.molecule.Molecule

Constructs a molecule object from a data structure.

Parameters

- **data** (*Union[str, Dict[str, Any], np.array]*) – Data to construct Molecule from
- **dtype** (*Optional[str], optional*) – How to interpret the data, if not passed attempts to discover this based on input type.
- **orient** (*bool, optional*) – Orientates the molecule to a standard frame or not.
- **validate** (*bool, optional*) – Validates the molecule or not.
- ****kwargs** (*Dict[str, Any]*) – Additional kwargs to pass to the constructors. kwargs take precedence over data.

Returns A constructed molecule class.

Return type *Molecule*

classmethod from_file(*filename: str, dtype: Optional[str] = None, *, orient: bool = False, **kwargs*)

Constructs a molecule object from a file.

Parameters

- **filename** (*str*) – The filename to build
- **dtype** (*Optional[str], optional*) – The type of file to interpret.
- **orient** (*bool, optional*) – Orientates the molecule to a standard frame or not.
- ****kwargs** – Any additional keywords to pass to the constructor

Returns A constructed molecule class.

Return type *Molecule*

get_fragment(*real: Union[int, List], ghost: Union[int, List, None] = None, orient: bool = False, group_fragments: bool = True*) → qcelemental.models.molecule.Molecule

Get new Molecule with fragments preserved, dropped, or ghosted.

Parameters

- **real** – Fragment index or list of indices (0-indexed) to be real atoms in new Molecule.

- **ghost** – Fragment index or list of indices (0-indexed) to be ghost atoms (basis fns only) in new Molecule.
- **orient** – Whether or not to align (inertial frame) and phase geometry upon new Molecule instantiation (according to _orient_molecule_internal)?
- **group_fragments** – Whether or not to group real fragments at the start of the atom list and ghost fragments toward the back. Previous to `v0.5`, this was always effectively True. True is handy for finding duplicate (atom-order-independent) molecules by hash. False preserves fragment order (though collapsing gaps for absent fragments) like Psi4's `extract_subsets`. False is handy for gradients where atom order of returned values matters.

Returns New `py::class:qcelemental.model.Molecule` with `self`'s fragments present, ghosted, or absent.

Return type mol

get_hash()
Returns the hash of the molecule.

get_molecular_formula()
Returns the molecular formula for a molecule. Atom symbols are sorted from A-Z.

Examples

```
>>> methane = qcelemental.models.Molecule('''
... H       0.5288       0.1610       0.9359
... C       0.0000       0.0000       0.0000
... H       0.2051       0.8240      -0.6786
... H       0.3345      -0.9314      -0.4496
... H      -1.0685      -0.0537       0.1921
... ''')
>>> methane.get_molecular_formula()
CH4
```

```
>>> hcl = qcelemental.models.Molecule('''
... H       0.0000       0.0000       0.0000
... Cl      0.0000       0.0000       1.2000
... ''')
>>> hcl.get_molecular_formula()
ClH
```

measure(*measurements: Union[List[int], List[List[int]]], *, degrees: bool = True*) → Union[float, List[float]]
Takes a measurement of the moleucle from the indicies provided.

Parameters

- **measurements** (*Union[List[int], List[List[int]]]*) – Either a single list of indices or multiple. Return a distance, angle, or dihedral depending if 2, 3, or 4 indices is provided, respectively. Values are returned in Bohr (distance) or degree.
- **degrees** (*bool, optional*) – Returns degrees by default, radians otherwise.

Returns Either a value or list of the measured values.

Return type Union[float, List[float]]

nelectrons(*ifr: int = None*) → int
Number of electrons.

Parameters **ifr** (*int, optional*) – If not *None*, only compute for the *ifr*-th (0-indexed) fragment.

Returns

Return type Number of electrons in entire molecule or in fragment.

nuclear_repulsion_energy(*ifr: int = None*) → float
Nuclear repulsion energy.

Parameters **ifr** (*int, optional*) – If not *None*, only compute for the *ifr*-th (0-indexed) fragment.

Returns

Return type Nuclear repulsion energy in entire molecule or in fragment.

orient_molecule()
Centers the molecule and orients via inertia tensor before returning a new Molecule

pretty_print()
Print the molecule in Angstroms. Same as `print_out()` only always in Angstroms. (method name in libmints is print_in_angstrom)

scramble(*, *do_shift: bool = True*, *do_rotate=True*, *do_resort=True*, *deflection=1.0*, *do_mirror=False*, *do_plot=False*, *do_test=False*, *run_to_completion=False*, *run_resorting=False*, *verbose=0*)
Generate a Molecule with random or directed translation, rotation, and atom shuffling. Optionally, check that the aligner returns the opposite transformation.

Parameters

- **ref_mol** (*qcel.models.Molecule*) – Molecule to perturb.
- **do_shift** (*bool or array-like, optional*) – Whether to generate a random atom shift on interval [-3, 3) in each dimension (*True*) or leave at current origin. To shift by a specified vector, supply a 3-element list.
- **do_rotate** (*bool or array-like, optional*) – Whether to generate a random 3D rotation according to algorithm of Arvo. To rotate by a specified matrix, supply a 9-element list of lists.
- **do_resort** (*bool or array-like, optional*) – Whether to shuffle atoms (*True*) or leave 1st atom 1st, etc. (*False*). To specify shuffle, supply a nat-element list of indices.
- **deflection** (*float, optional*) – If *do_rotate*, how random a rotation: 0.0 is no change, 0.1 is small perturbation, 1.0 is completely random.
- **do_mirror** (*bool, optional*) – Whether to construct the mirror image structure by inverting y-axis.
- **do_plot** (*bool, optional*) – Pops up a mpl plot showing before, after, and ref geometries.
- **do_test** (*bool, optional*) – Additionally, run the aligner on the returned Molecule and check that opposite transformations obtained.
- **run_to_completion** (*bool, optional*) – By construction, scrambled systems are fully alignable (final RMSD=0). Even so, *True* turns off the mechanism to stop when RMSD reaches zero and instead proceed to worst possible time.

- **run_resorting** (*bool, optional*) – Even if atoms not shuffled, test the resorting machinery.
- **verbose** (*int, optional*) – Print level.

Returns Molecule is scrambled copy of *ref_mol* (self). *data['rmsd']* is RMSD [A] between *ref_mol* and the scrambled geometry. *data['mill']* is a AlignmentMill with fields (shift, rotation, atommap, mirror) that prescribe the transformation from *ref_mol* to the returned geometry.

Return type *Molecule*, data

Raises **AssertionError** – If *do_test=True* and aligner sanity check fails for any of the reverse transformations.

show(*ngl_kwargs: Optional[Dict[str, Any]] = None*) → nglview.NGLWidget
Creates a 3D representation of a moleucle that can be manipulated in Jupyter Notebooks and exported as images (*.png*).

Parameters **ngl_kwargs** (*Optional[Dict[str, Any]], optional*) – Addition nglview NGLWidget kwargs

Returns A nglview view of the molecule

Return type nglview.NGLWidget

to_file(*filename: str, dtype: Optional[str] = None*) → None
Writes the Molecule to a file.

Parameters

- **filename** (*str*) – The filename to write to
- **dtype** (*Optional[str], optional*) – The type of file to write, attempts to infer dtype from the filename if not provided.

to_string(*dtype: str, units: str = None, *, atom_format: str = None, ghost_format: str = None, width: int = 17, prec: int = 12, return_data: bool = False*)
Returns a string that can be used by a variety of programs.

Unclear if this will be removed or renamed to "to_psi4_string" in the future

Suggest psi4 –> psi4frag and psi4 route to to_string

4.8 AtomicResult

A Python implementation of the MolSSI QCSchema `AtomicResult` object.

4.8.1 AtomicInput

class qcelemental.models.**AtomicInput**
The MolSSI Quantum Chemistry Schema

Parameters

- **id** (*str, Optional*) – An optional ID of the ResultInput object.
- **schema_name** (*ConstrainedStrValue, Default: qcschema_input*) –
- **schema_version** (*int, Default: 1*) –
- **molecule** (*Molecule*) – The molecule to use in the computation.

- **driver** (*{energy,gradient,hessian,properties}*) – Allowed quantum chemistry driver values.
- **model** (Model) – The quantum chemistry model specification for a given operation to compute against
- **keywords** (*Dict[str, Any], Default: {}*) – The program specific keywords to be used.
- **protocols** (AtomicResultProtocols, Optional) – Protocols regarding the manipulation of a Result output data.
- **extras** (*Dict[str, Any], Default: {}*) – Extra fields that are not part of the schema.
- **provenance** (*Provenance*, Optional) – Provenance information.

4.8.2 AtomicResult

class qcelemental.models.**AtomicResult**

Parameters

- **id** (*str, Optional*) – An optional ID of the ResultInput object.
- **schema_name** (*ConstrainedStrValue, Default: qcschema_output*) –
- **schema_version** (*int, Default: 1*) –
- **molecule** (*Molecule*) – The molecule to use in the computation.
- **driver** (*{energy,gradient,hessian,properties}*) – Allowed quantum chemistry driver values.
- **model** (Model) – The quantum chemistry model specification for a given operation to compute against
- **keywords** (*Dict[str, Any], Default: {}*) – The program specific keywords to be used.
- **protocols** (AtomicResultProtocols, Optional) – Protocols regarding the manipulation of a Result output data.
- **extras** (*Dict[str, Any], Default: {}*) – Extra fields that are not part of the schema.
- **provenance** (*Provenance*) – Provenance information.
- **properties** (AtomicResultProperties) – Named properties of quantum chemistry computations following the MolSSI QCSchema.
- **wavefunction** (WavefunctionProperties, Optional) – None
- **return_result** (*Union[float, Array, Dict[str, Any]]*) – The value requested by the 'driver' attribute.
- **stdout** (*str, Optional*) – The standard output of the program.
- **stderr** (*str, Optional*) – The standard error of the program.
- **success** (*bool*) – The success of a given programs execution. If False, other fields may be blank.
- **error** (*ComputeError*, Optional) – A complete description of the error.

4.8.3 API

class qcelemental.models.results.**AtomicResultProtocols**
Protocols regarding the manipulation of a Result output data.

Parameters

- **wavefunction** (*{all,orbitals_and_eigenvalues,return_results, none}, Default: none*) – Wavefunction to keep from a Result computation.
- **stdout** (*bool, Default: True*) – Primary output file to keep from a Result computation

class qcelemental.models.results.**AtomicResultProperties**
Named properties of quantum chemistry computations following the MolSSI QCSchema.

Parameters

- **calcinfo_nbasis** (*int, Optional*) – The number of basis functions for the computation.
- **calcinfo_nmo** (*int, Optional*) – The number of molecular orbitals for the computation.
- **calcinfo_nalpha** (*int, Optional*) – The number of alpha electrons in the computation.
- **calcinfo_nbeta** (*int, Optional*) – The number of beta electrons in the computation.
- **calcinfo_natom** (*int, Optional*) – The number of atoms in the computation.
- **nuclear_repulsion_energy** (*float, Optional*) – The nuclear repulsion energy energy.
- **return_energy** (*float, Optional*) – The energy of the requested method, identical to *return_value* for energy computations.
- **scf_one_electron_energy** (*float, Optional*) – The one-electron (core Hamiltonian) energy contribution to the total SCF energy.
- **scf_two_electron_energy** (*float, Optional*) – The two-electron energy contribution to the total SCF energy.
- **scf_vv10_energy** (*float, Optional*) – The VV10 functional energy contribution to the total SCF energy.
- **scf_xc_energy** (*float, Optional*) – The functional (XC) energy contribution to the total SCF energy.
- **scf_dispersion_correction_energy** (*float, Optional*) – The dispersion correction appended to an underlying functional when a DFT-D method is requested.
- **scf_dipole_moment** (*List[float], Optional*) – The X, Y, and Z dipole components.
- **scf_total_energy** (*float, Optional*) – The total electronic energy of the SCF stage of the calculation.
- **scf_iterations** (*int, Optional*) – The number of SCF iterations taken before convergence.
- **mp2_same_spin_correlation_energy** (*float, Optional*) – The portion of MP2 doubles correlation energy from same-spin (i.e. triplet) correlations.

- **mp2_opposite_spin_correlation_energy** (*float, Optional*) – The portion of MP2 doubles correlation energy from opposite-spin (i.e. singlet) correlations.
- **mp2_singles_energy** (*float, Optional*) – The singles portion of the MP2 correlation energy. Zero except in ROHF.
- **mp2_doubles_energy** (*float, Optional*) – The doubles portion of the MP2 correlation energy including same-spin and opposite-spin correlations.
- **mp2_total_correlation_energy** (*float, Optional*) – The MP2 correlation energy.
- **mp2_correlation_energy** (*float, Optional*) – The MP2 correlation energy.
- **mp2_total_energy** (*float, Optional*) – The total MP2 energy (MP2 correlation energy + HF energy).
- **mp2_dipole_moment** (*List[float], Optional*) – The MP2 X, Y, and Z dipole components.
- **ccsd_same_spin_correlation_energy** (*float, Optional*) – The portion of CCSD doubles correlation energy from same-spin (i.e. triplet) correlations.
- **ccsd_opposite_spin_correlation_energy** (*float, Optional*) – The portion of CCSD doubles correlation energy from opposite-spin (i.e. singlet) correlations
- **ccsd_singles_energy** (*float, Optional*) – The singles portion of the CCSD correlation energy. Zero except in ROHF.
- **ccsd_doubles_energy** (*float, Optional*) – The doubles portion of the CCSD correlation energy including same-spin and opposite-spin correlations.
- **ccsd_correlation_energy** (*float, Optional*) – The CCSD correlation energy.
- **ccsd_total_energy** (*float, Optional*) – The total CCSD energy (CCSD correlation energy + HF energy).
- **ccsd_dipole_moment** (*List[float], Optional*) – The CCSD X, Y, and Z dipole components.
- **ccsd_iterations** (*int, Optional*) – The number of CCSD iterations taken before convergence.
- **ccsd_prt_pr_correlation_energy** (*float, Optional*) – The CCSD(T) correlation energy.
- **ccsd_prt_pr_total_energy** (*float, Optional*) – The total CCSD(T) energy (CCSD(T) correlation energy + HF energy).
- **ccsd_prt_pr_dipole_moment** (*List[float], Optional*) – The CCSD(T) X, Y, and Z dipole components.

class qcelemental.models.results.**WavefunctionProperties**

Parameters **basis** – A quantum chemistry basis description.

restricted [bool] If the computation was restricted or not (alpha == beta). If True, all beta quantities are skipped.

h_core_a [Array, Optional] Alpha-spin core (one-electron) Hamiltonian.

h_core_b [Array, Optional] Beta-spin core (one-electron) Hamiltonian.

h_effective_a [Array, Optional] Alpha-spin effective core (one-electron) Hamiltonian.

h_effective_b [Array, Optional] Beta-spin effective core (one-electron) Hamiltonian

scf_orbitals_a [Array, Optional] SCF alpha-spin orbitals.

scf_orbitals_b [Array, Optional] SCF beta-spin orbitals.

scf_density_a [Array, Optional] SCF alpha-spin density matrix.

scf_density_b [Array, Optional] SCF beta-spin density matrix.

scf_fock_a [Array, Optional] SCF alpha-spin Fock matrix.

scf_fock_b [Array, Optional] SCF beta-spin Fock matrix.

scf_eigenvalues_a [Array, Optional] SCF alpha-spin eigenvalues.

scf_eigenvalues_b [Array, Optional] SCF beta-spin eigenvalues.

scf_occupations_a [Array, Optional] SCF alpha-spin occupations.

scf_occupations_b [Array, Optional] SCF beta-spin occupations.

orbitals_a [str, Optional] Index to the alpha-spin orbitals of the primary return.

orbitals_b [str, Optional] Index to the beta-spin orbitals of the primary return.

density_a [str, Optional] Index to the alpha-spin density of the primary return.

density_b [str, Optional] Index to the beta-spin density of the primary return.

fock_a [str, Optional] Index to the alpha-spin Fock matrix of the primary return.

fock_b [str, Optional] Index to the beta-spin Fock matrix of the primary return.

eigenvalues_a [str, Optional] Index to the alpha-spin eigenvalues of the primary return.

eigenvalues_b [str, Optional] Index to the beta-spin eigenvalues of the primary return.

occupations_a [str, Optional] Index to the alpha-spin orbital eigenvalues of the primary return.

occupations_b [str, Optional] Index to the beta-spin orbital eigenvalues of the primary return.

4.9 Common

Common Models used throughout the QCArchive ecosystem.

4.9.1 BasisSet

class qcelemental.models.**BasisSet**

A quantum chemistry basis description.

Parameters

- **schema_name** (*ConstrainedStrValue, Default: qcschema_basis*) –
- **schema_version** (*int, Default: 1*) –
- **name** (*str*) – A standard basis name if available (e.g., 'cc-pVDZ').
- **description** (*str, Optional*) – A brief description of the basis set.
- **center_data** (BasisCenter) – A mapping of all types of centers available.
- **atom_map** (*List[str]*) – Mapping of all centers in the parent molecule to centers in *center_data*.
- **nbf** (*int, Optional*) – The number of basis functions.

class qcelemental.models.basis.**BasisCenter**
Data for a single atom/center in a basis set.

Parameters

- **electron_shells** (*ElectronShell*) – Electronic shells for this center.
- **ecp_electrons** (*int, Default: 0*) – Number of electrons replace by ECP potentials.
- **ecp_potentials** (*ECPPotential*, Optional) – ECPs for this center.

class qcelemental.models.basis.**ElectronShell**
Information for a single electronic shell

Parameters

- **angular_momentum** (*List[int]*) – Angular momentum for this shell.
- **harmonic_type** (*{spherical,cartesian}*) – The angular momentum representation of a shell.
- **exponents** (*List[float]*) – Exponents for this contracted shell.
- **coefficients** (*List[List[float]]*) – General contraction coefficients for this shell, individual list components will be the individual segment contraction coefficients.

class qcelemental.models.basis.**ECPPotential**
Information for a single ECP potential.

Parameters

- **ecp_type** (*{scalar,spinorbit}*) – The type of the ECP potential.
- **angular_momentum** (*List[int]*) – Angular momentum for the ECPs.
- **r_exponents** (*List[int]*) – Exponents of the 'r' term.
- **gaussian_exponents** (*List[float]*) – Exponents of the 'gaussian' term.
- **coefficients** (*List[List[float]]*) – General contraction coefficients for this shell, individual list components will be the individual segment contraction coefficients.

4.9.2 ComputeError

class qcelemental.models.**ComputeError**
A complete description of the error.

Parameters

- **error_type** (*str*) – The type of error which was thrown. Restrict this field short classifiers e.g. 'input_error'.
- **error_message** (*str*) – Text associated with the thrown error, often the backtrace, but can contain additional information as well.
- **extras** (*Dict[str, Any], Optional*) – Additional data to ship with the ComputeError object.

4.9.3 FailedOperation

class qcelemental.models.**FailedOperation**

A record indicating that a given operation (compute, procedure, etc.) has failed and contains the reason and input data which generated the failure.

Parameters

- **id** (*str, Optional*) – A unique identifier which links this FailedOperation, often of the same Id of the operation should it have been successful. This will often be set programmatically by a database such as Fractal.
- **input_data** (*Any, Optional*) – The input data which was passed in that generated this failure. This should be the complete input which when attempted to be run, caused the operation to fail.
- **success** (*bool, Default: False*) – A boolean indicator that the operation failed consistent with the model of successful operations. Should always be False. Allows programmatic assessment of all operations regardless of if they failed or succeeded
- **error** (*ComputeError*) – A container which has details of the error that failed this operation. See the *ComputeError* for more details.
- **extras** (*Dict[str, Any], Optional*) – Additional information to bundle with this Failed Operation. Details which pertain specifically to a thrown error should be contained in the *error* field. See *ComputeError* for details.

4.9.4 Provenance

class qcelemental.models.**Provenance**

Provenance information.

Parameters

- **creator** (*str*) – The creator of the object.
- **version** (*str, Optional*) – The version of the creator.
- **routine** (*str, Optional*) – The routine of the creator.

Developer Documentation

Contains in-depth developer documentation and API references.

- *QCElemental API*
- *Changelog*

4.10 QCElemental API

4.10.1 qcelemental Package

Main init for QCElemental

Functions

compare(expected, computed[, label, quiet, ...])	Returns True if two integers, strings, booleans, or integer arrays are element-wise equal.
compare_recursive(expected, computed[, ...])	Recursively compares nested structures such as dictionaries and lists.
compare_values(expected, computed[, label, ...])	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: Union[Dict, pydantic.main.BaseModel, ProtoModel]*, *computed: Union[Dict, pydantic.main.BaseModel, ProtoModel]*, *label: str = None*, ***, *atol: float = 1e-06*, *rtol: float = 1e-16*, *forgive: List[str] = 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

ChoicesError(msg[, choices])	Error called for problems with syntax input file.
CovalentRadii([context])	Covalent radii sets.
DataUnavailableError(dataset, atom)	Error when dataset incomplete and otherwise valid query can't be fulfilled.
Datum(label, units, data, *[, comment, doi, ...])	Facilitates the storage of quantum chemical results by labeling them with basic metadata.
MoleculeFormatError(msg)	Error called when a molparse.from_string contains unparsable lines.
NotAnElementError(atom)	Error when element or nuclide can't be identified.
PhysicalConstantsContext([context])	CODATA 2014 physical constants set from NIST.
ValidationError(msg)	Error called for problems with syntax input file.
VanderWaalsRadii([context])	Van der Waals radii sets.

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

get(atom, *[, return_tuple, units, missing])	Access a covalent radius for species *atom*.
string_representation()	Print name, value, and units of all covalent radii.
write_c_header([filename, missing])	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:
    rad = qcel.covalentradii.get(atom)
except qcel.DataUnavailableError:
    rad = 4.0
```

  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.

- *qcelemental.Datum* – When `return_tuple=True`, Datum 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 qcelemental.**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

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

Methods Summary

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

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']

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

VanderWaalsRadii

class qcelemental.**VanderWaalsRadii**(*context: str = 'MANTINA2009'*)
Bases: object

Van der Waals radii sets.

Parameters **context** (*{'MANTINA2009'}*) – Origin of loaded data.

vdwr
Each van der Waals radius is an entry in *vdwr*, 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* and *data* value as Decimal object.

Type dict of Datum

doi
The DOI of the current context.

Type str

name
The name of the context ('MANTINA2009')

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

get(atom, *[, return_tuple, units, missing])	Access a van der Waals radius for species `atom`.
string_representation()	Print name, value, and units of all van der Waals radii.
write_c_header([filename, missing])	Write C header file defining Van der Waals 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 van der Waals radius for species `atom`.

Parameters

- **atom** (*int or str*) – Identifier for element or nuclide, e.g., `H`, `C`, `Al`.
- **units** (*str, optional*) – Units of returned value. To return in native unit (MANTINA2009: 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:
    rad = qcel.vdwradii.get(atom)
except qcel.DataUnavailableError:
    rad = 4.0
```

  Only relevant for `return_tuple=False`.
- **return_tuple** (*bool, optional*) – See below.

Returns

- *float* – When `return_tuple=False`, value of Van der Waals radius. If multiple defined for element, returns largest.
- *qcelemental.Datum* – When `return_tuple=True`, Datum with units, description, uncertainty, and value of van der Waals 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 van der Waals radius is available and *missing=None*.

string_representation() → str
Print name, value, and units of all van der Waals radii.

write_c_header(*filename: str = 'vdwrad.h'*, *missing: float = 2.0*) → None
Write C header file defining Van der Waals 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.

Variables

constants	CODATA 2014 physical constants set from NIST.
periodictable	Nuclear and mass data about chemical elements from NIST.
vdwradii	Van der Waals radii sets.

constants

qcelemental.**constants = <qcelemental.physical_constants.context.PhysicalConstantsContext ob**
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

qcelemental.**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

qcelemental.**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

qcelemental.**name**

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

Type list of str

vdwradii

qcelemental.**vdwradii = <qcelemental.vanderwaals_radii.VanderWaalsRadii object>**

Van der Waals radii sets.

Parameters **context** (*{'MANTINA2009'}*) – Origin of loaded data.

qcelemental.**vdwr**

Each van der Waals radius is an entry in *vdwr*, 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* and *data* value as Decimal object.

Type dict of Datum

qcelemental.**doi**

The DOI of the current context.

Type str

qcelemental.**name**

The name of the context ('MANTINA2009')

Type str

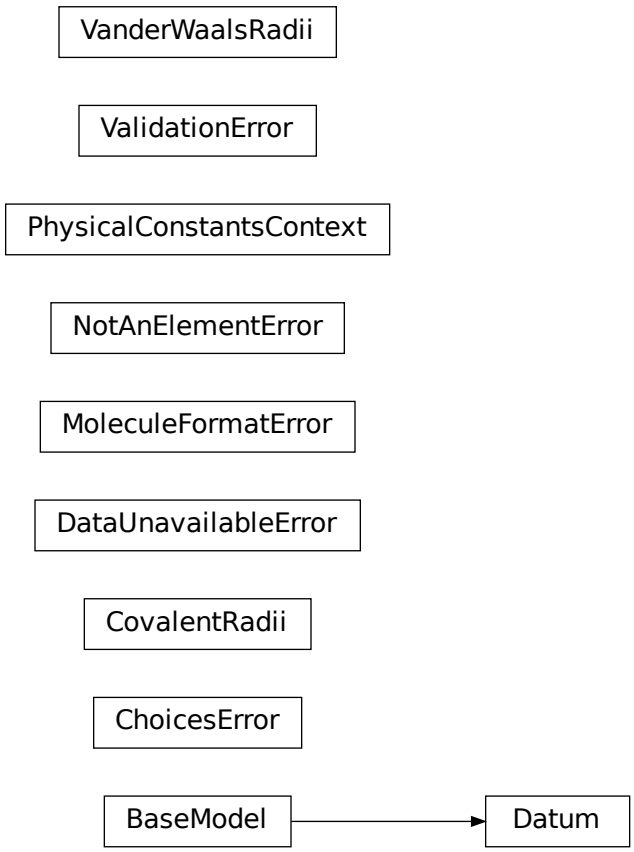
qcelemental.**native_units**
The units the original data was provided in.

Type str

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

Type int

Class Inheritance Diagram



4.10.2 qcelemental.periodic_table Module

Periodic table class

Classes

PeriodicTable()	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

run_comparison()	Compare the existing element information for Psi4 and Cfour (in checkup_data folder) to *self*.
to_A(atom)	Get mass number of *atom*.
to_E(atom)	Get element symbol of *atom*.
to_Z(atom)	Get atomic number of *atom*.
to_atomic_number(atom)	Get atomic number of *atom*.
to_element(atom)	Get element name of *atom*.
to_group(atom)	Get group (vertical column in periodic table) of *atom*.
to_mass(atom, *[, return_decimal])	Get atomic mass of *atom*.

Continued on next page

Table 13 – continued from previous page

to_mass_number(atom)	Get mass number of *atom*.
to_name(atom)	Get element name of *atom*.
to_period(atom)	Get period (horizontal row in periodic table) of *atom*.
to_symbol(atom)	Get element symbol of *atom*.
write_c_header([filename])	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

4.10.3 qcelemental.physical_constants Package

Units handlers

Classes

PhysicalConstantsContext([context])	CODATA 2014 physical constants set from NIST.

PhysicalConstantsContext

class qcelemental.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

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

Methods Summary

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

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']

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

4.10.4 qcelemental.covalent_radii Module

Contains covalent radii

Classes

CovalentRadii([context])	Covalent radii sets.

CovalentRadii

class qcelemental.covalent_radii.**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

get(atom, *[, return_tuple, units, missing])	Access a covalent radius for species *atom*.
string_representation()	Print name, value, and units of all covalent radii.
write_c_header([filename, missing])	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:
    rad = qcel.covalentradii.get(atom)
except qcel.DataUnavailableError:
    rad = 4.0
```

  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.
- *qcelemental.Datum* – When `return_tuple=True`, Datum 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.

Class Inheritance Diagram

4.10.5 qcelemental.molparse Package

Functions

contiguize_from_fragment_pattern(frag_pattern, *)	Take (nat, ?) array-like arrays and return with atoms arranged by (nfr, ?) *frag_pattern*.
from_arrays(*[, geom, elea, elez, elem, ...])	Compose a Molecule dict from unvalidated arrays and variables, returning dict.
from_input_arrays(*[, enable_qm, ...])	Compose a Molecule dict from unvalidated arrays and variables in multiple domains.
from_schema(molschema, *[, verbose])	Construct molecule dictionary representation from non-Psi4 schema.
from_string(molstr[, dtype, name, fix_com, ...])	Construct a molecule dictionary from any recognized string format.
parse_nucleus_label(label)	Separate molecule nucleus string into fields.
reconcile_nucleus	Forms consistent set of nucleus descriptors from all information from arguments, supplemented by the periodic table.
to_schema(molrec, dtype[, units, np_out, copy])	Translate molparse internal Molecule spec into dictionary from other schemas.
to_string(molrec, dtype[, units, ...])	Format a string representation of QM molecule.
validate_and_fill_chgmult(zeff, ...[, ...])	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 ***qcelemental.ValidationError*** – When *frag_pattern* skips atoms or any array has inconsistent length. If *throw_reorder*, raises when non-contiguous fragments detected.

from_arrays

qcelemental.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*, *fragment_separators=None*, *fragment_charges=None*, *fragment_multiplicities=None*, *molecular_charge=None*, *molecular_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 *qcelemental.ValidationError* – For most anything wrong.

from_input_arrays

qcelemental.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 *qcelemental.molparse.from_arrays()* for sucessive domains and hooks them together (e.g., impose *fix_com* on "qm" when "efp" present.

from_schema

qcelemental.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

qcelemental.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
    ---------
    Specifiable: 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
    ---------
    Specifiable: 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

qcelemental.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: Union[str, int], 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*) → Union[str, Tuple[str, Dict]]

Format a string representation of QM molecule.

Parameters

- **molrec** (*Dict*) – Psi4 json Molecule spec.
- **dtype** (*str, {'xyz', 'cfour', 'nwchem', 'molpro', 'turbomole', 'qchem'}*) – Overall string format. Note that it's possible to request variations that don't fit the dtype spec so may not be re-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.

- **return_data** (*bool, optional*) – 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 `qcelemental.models.Molecule` expressed into

  string _or_ keywords.

validate_and_fill_chgmult

qcelemental.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 constained 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: mult - 1 <= neut_el - chg
  - R5 require total parity consistent among tot electrons and mult: (mult % 2) != ((neut_el - chg) % 2)
  - R6 require chg match input argument values
  - R7 require mult match input argument values
  - R8 require that tot = sum(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 = tot - high_spin_sum(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]))}
```

4.10.6 qcelemental.testing Module

Functions

compare(expected, computed[, label, quiet, ...])	Returns True if two integers, strings, booleans, or integer arrays are element-wise equal.
compare_molrecs(expected, computed[, label, ...])	Function to compare Molecule dictionaries.
compare_recursive(expected, computed[, ...])	Recursively compares nested structures such as dictionaries and lists.
compare_values(expected, computed[, label, ...])	Returns True if two floats or float arrays are element-wise equal within a tolerance.
tnm()	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: Union[Dict, pydantic.main.BaseModel, ProtoModel]*, *computed: Union[Dict, pydantic.main.BaseModel, ProtoModel]*, *label: str = None*, ***, *atol: float = 1e-06*, *rtol: float = 1e-16*, *forgive: List[str] = 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.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:

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

tnm

qcelemental.testing.**tnm**() → str

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

4.11 Changelog

4.11.1 0.12.0 / 2019-11-13

New Features

Enhancements

- (GH#156) `Molecules` can now be correctly compared with ==.
- (GH#157) `molparse.to_string` Q-Chem dtype developed. Psi4 dtype now includes label and doesn't have extraneous info for single fragment systems.
- (GH#162) New protocol `stdout` added to `ResultProtocols` controlling whether `stdout` field (which generally contains the primary logfile, whether a program writes it to file or stdout) is returned.
- (GH#165) The code base is now Black formatted.

Deprecations

- (GH#156) `Molecule.compare` is deprecated and will be removed in v0.13.0.
- (GH#167, GH#168) `ResultInput`, `Result`, `Optimization` have been removed in favor of `AtomicInput`, `AtomicResult`, and `OptimizationResult` and will be removed in v0.13.0.

Bug Fixes

- (GH#170) `ProtoModel` subclasses now correctly allow custom `__repr__` and `__str__` methods.
- (GH#164, GH#166) `nglview-sdf` molecule string format now correctly uses correct sdf format widths fixing some issues with very large molecules.

4.11.2 0.11.1 / 2019-10-28

Bug Fixes

- **(GH#152) Patches `Molecule.from_file` and `Molecule.from_data` to read XYZ+ format and correctly handle key**
  Patches `Molecule.to_file` to write XYZ+ format as the default for XYZ and XYZ+ files.

4.11.3 0.11.0 / 2019-10-24

Enhancements

- (GH#147) Updates Pydantic to the 1.0 release and fixes a number of breaking changes.
- (GH#148) Switches from Py3dMoljs to NGLView for molecular visualization due to Jupyter Widget integration.
- (GH#149) Adds statC and Debye to the units registry.

Bug Fixes

- (GH#150) Patches `which_import` to correctly handle submodules.

4.11.4 0.10.0 / 2019-10-16

Enhancements

- (GH#144) Allows *which_import* to handle submodules.
- (GH#143) Allow testing complex numbers.

4.11.5 0.9.0 / 2019-10-01

New Features

- (GH#137, GH#138) Coordinates can now be output in `Turbomole` format in addition to all other formats.
- (GH#139) A wavefunction property have been added to the `Result` Model. Adds the ability for Engine and other programs to store and fetch wavefunction data.
- (GH#140) `Protocols` have been added to `QCInputSpecification` which allows data to pre-pruned by different specifications. Main intention is to reduce wavefunction data which may be re-computed cheaply rather than storing all of it. This does change the input model, so requires a minor version bump.

Enhancements

- (GH#132) `BasisSet` and `Result`'s documentations have been brought up to the standards of other models.

4.11.6 0.8.0 / 2019-09-13

New Features

- (GH#123) QCElemental now passes MyPy!
- (GH#127, GH#131) Adds van der Waals radii data available through `vdwradii.get(atom)` function.

Enhancements

Bug Fixes

- (GH#125) Add back a consistency check that had been optimized out.

4.11.7 0.7.0 / 2019-08-23

Enhancements

- (GH#118) Model string representations should be more user friendly and descriptive without overload the output.
- (GH#119) The `molparse.to_string` keyword-arg `return_data` now returns molecule keywords for GAMESS and NWChem. The `models.Molecule.to_string` can use `return_data` now, too.
- (GH#120) Auto documentation tech is now built into the `ProtoModel` and does not need an external function.

4.11.8 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.

4.11.9 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.

4.11.10 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`.

4.11.11 0.4.2 / 2019-06-13

New Features

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

4.11.12 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. `Datum`s 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`.

4.11.13 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.

4.11.14 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.

4.11.15 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.

4.11.16 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

4.11.17 0.3.0 / 2019-02-27

New Features

- (GH#33) `molparse.to_schema` recognizes `dtype=2` in keeping with GH:MolSSI/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

4.11.18 0.2.6 / 2019-02-18

Bug Fixes

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

4.11.19 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.

4.11.20 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.

4.11.21 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.

4.11.22 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.

4.11.23 0.2.1 / 2019-01-27

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

4.11.24 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.

4.11.25 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.

4.11.26 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.

4.11.27 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.

4.11.28 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:MolSSI/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)`.

4.12 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

INDEX

G

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z