PyAvia

Release 0.0.3

Jan 24, 2022

Contents

PyAvia is a collection of modules useful for common tasks in aerospace engineering or engineering in general. Some things to note:

- In code snippets, it may be assumed that PyAvia has been imported as follows:

```
>>> import pyavia as pa
```

- Code examples are shown in the repository under `examples/`.

Warning: *CAVEAT COMPUTOR* - These modules are provided "as is", without warranty of any kind. They are intended to be read and/or used by people trained in engineering and scientific methods who know how to verify results and who can recognise incorrect values when they see them... *which will happen frequently.*

CHAPTER 1

Version

The current version is **0.0.3**. **PyAvia** is designed for Python >= 3.9 and is platform agnostic.

Note: At this stage PyAvia is extremely preliminary, alpha, pre-release, etc. Structural changes may be made to the code at any time that will almost definitely break third party code. Please don't get cross.

CHAPTER 2

Packages

2.1 Aerodynamics Modules

2.1.1 Contents

Atmosphere

pyavia.aero.Atmosphere
pyavia.aero.geo_alt_to_pot
pyavia.aero.pot_alt_to_geo

General Gases

pyavia.aero.Gas

Perfect Gases

pyavia.aero.PerfectGas
pyavia.aero.perfect_gas.enthalpy
pyavia.aero.perfect_gas. stag_press_ratio
pyavia.aero.perfect_gas. stag_temp_ratio
pyavia.aero.perfect_gas. temp_from_enthalpy

Imperfect Gases

pyavia.aero.ImperfectGas

2.1.2 Members

2.2 Containers

2.2.1 Contents

pyavia.containers.AttrDict	AttrDict is a dictionary class that also allows access using attribute notation.
pyavia.containers.MultiBiDict	Bi-directional dict class.
pyavia.containers.ValueRange	Represents any range of scalar values (units agnostic).
pyavia.containers.wdg_edge	alias of `builtins.slice`
pyavia.containers.WtDirGraph	A weighted directed graph where a value / weight can be assigned to any link / edge between any two hashable nodes / keys.

2.2.2 Members

Adds useful, less common containers not available in the standard library.

class pyavia.containers.**AttrDict**
Bases: `dict`

AttrDict is a dictionary class that also allows access using attribute notation.

Examples

```
>>> my_dict = AttrDict([('make', 'Mazda'), ('model', '3')])
>>> my_dict['yom'] = 2007
>>> print(f"My car is a {my_dict.make} {my_dict.model} made in "
...       f"{my_dict.yom}.")
My car is a Mazda 3 made in 2007.
```

__delattr__
Delete self[key].

__dir__()
Default dir() implementation.

__getattr__(*name*)

__repr__()
Return repr(self).

__setattr__
Set self[key] to value.

class pyavia.containers.**MultiBiDict**(**args*, ***kwargs*)
Bases: `dict`

Bi-directional dict class.

A forward and inverse dictionary are synchronised to allow searching by either key or value to get the corresponding value / key. Implementation from this StackOverflow answer: https://stackoverflow.com/a/21894086.

Notes

- The inverse dict bidict_multi.inverse auto-updates itself when the normal dict bidict_multi is modified.
- Inverse directory entries bidict_multi.inverse[value] are always lists of key such that bidict_multi[key] == value.
- Multiple keys can have the same value.

Examples

```
>>> bd = MultiBiDict({'a': 1, 'b': 2})
>>> print(bd)
{'a': 1, 'b': 2}
>>> print(bd.inverse)
{1: ['a'], 2: ['b']}
>>> bd['c'] = 1          # Now two keys have the same value (= 1)
>>> print(bd)
{'a': 1, 'b': 2, 'c': 1}
>>> print(bd.inverse)
{1: ['a', 'c'], 2: ['b']}
>>> del bd['c']
>>> print(bd)
{'a': 1, 'b': 2}
>>> print(bd.inverse)
{1: ['a'], 2: ['b']}
>>> del bd['a']
>>> print(bd)
{'b': 2}
>>> print(bd.inverse)
{2: ['b']}
>>> bd['b'] = 3
>>> print(bd)
{'b': 3}
>>> print(bd.inverse)
{2: [], 3: ['b']}
```

__delitem__(*key*)
Delete self[key].

__init__(**args*, ***kwargs*)

Parameters **args,kwargs** (*dict, optional*) – Initialise the forward dict as dict(**args*, ***kwargs*) if supplied. The inverse dict is then constructed.

__setitem__(*key*, *value*)
Set self[key] to value.

class pyavia.containers.**ValueRange**(**, x_min=None, x_max=None, x_mean=None, x_ampl=None*)

Bases: object

Represents any range of scalar values (units agnostic).

__init__(*, *x_min=None*, *x_max=None*, *x_mean=None*, *x_ampl=None*)
Establish a range using any valid combination of two of the input arguments from *x_min*, *x_max*, *x_mean*, *x_ampl*.

Note: If minimum and maximum are reversed (or amplitude is negative) these will be automatically switched which may produce unexpected results.

Parameters

- **x_min** (*scalar*) – Range minimum.
- **x_max** (*scalar*) – Range maximum.
- **x_mean** (*scalar*) – Range mean.
- **x_ampl** (*scalar*) – Range amplitude of variation (i.e. +/- value to superimpose on mean).

ampl
Computed amplitude.

mean
Computed mean value.

class pyavia.containers.**WtDirGraph**(*links: Dict[Hashable, Dict[Hashable, Any]] = None*)
Bases: object

A weighted directed graph where a value / weight can be assigned to any link / edge between any two hashable nodes / keys. Link values can be different in each direction (or completely absent), i.e. x →[value]→ y and x ←[reverse_value2]← y. Intended for sparse graphs. Implemented as a dict(x-keys)-of-dicts(y-keys), both forward and reverse values are therefore stored independently for each pair of keys.

Access to all links from a given key is the same as a dict, but access to a specific link between two keys uses the slice [:] notation. e.g.: `wdg['a':'b']` returns the value between $a \rightarrow b$, whereas `wdg['a']` returns a dict of all link / edge values starting at *a* i.e. `{'b': 'somevalue', 'c': 'anotherval'}`.

Note: Where slice / square bracket syntax cannot be used, alias wdg_edge(a, b) is defined as an alias of the builtin slice function. This is used to define a graph edge. e.g.: `if wdg_edge(a, b) in wdg:`.

Examples

```
>>> wdg = pa.WtDirGraph()
>>> wdg['a':'b'] = 'Here'
>>> print(wdg)
WtDirgraph({'a': {'b': 'Here'}, 'b': {}})
>>> print(wdg['a':'b'])
Here
>>> print(wdg['b':'a'])
Traceback (most recent call last):
...
KeyError: 'a'
>>> wdg['a':3.14159] = (22, 7)
>>> wdg['b':'c'] = 'it'
>>> wdg['c':'d'] = 'is.'
```

(continues on next page)

(continued from previous page)

```
>>> path, joined = wdg.trace('a', 'd', op=lambda x, y: ' '.join([x, y]))
>>> print(path, joined)
['a', 'b', 'c', 'd'] Here it is.
>>> del wdg['a']
>>> print(wdg)
WtDirgraph({'b': {'c': 'it'}, 3.14159: {}, 'c': {'d': 'is.'}, 'd': {}})
```

__contains__(*arg*)
Returns True if a link value x → y exists (slice arg), or True if key exists (single arg). Because standalone slice notation is not available on the LHS, use the following syntax: `wdg_edge(x, y) in wdg`.

Parameters **arg** (*slice or key*) – If slice, [from:to] otherwise key value *x*.

Returns

result –

- Slice arg: True if a link value *x* → *y* exists, else False.
- Key arg: True if key exists (single arg), else False.

Return type bool

__delitem__(*arg*)
Delete link value for x → y (slice arg) or delete key *x* and all associated links (single arg).

Parameters **arg** (*slice or key*) – If slice, [from:to] otherwise key value *x*.

Raises `KeyError` – Invalid slice argument or if link/s do not exist.

__getitem__(*arg: slice*)
Get link value for x → y.

Parameters **arg** (*slice*) – [from:to].

Returns Link value.

Raises `KeyError` – Invalid slice argument or link does not exist.

__init__(*links: Dict[Hashable, Dict[Hashable, Any]] = None*)
Construct a weighted directional graph.

Parameters **links** – If provided this is to be a dict-of-dicts representing forward links. Each key corresponds to a dict holding other keys and the link value. Example:

```
>>> wdg = WtDirGraph({'a': {'b': 2, 'c': 5}, 'c': {'a': 4}})
>>> print(wdg['c':'a'])
4
```

Creates a graph with linkages of:

```
a -→ b  link value = 2
  |→ c  link value = 5
b -→    (no links)
c -→ a  link value = 4
```

__repr__()
Return repr(self).

__setitem__(*arg: slice*, *value*)
Set / overwrite link value for x → y.

Parameters

- **arg** (*slice*) – [from:to]
- **value** – Link value.

Raises KeyError – Invalid slice argument (including path to nowhere *x* == *y*).

trace(*x: Hashable, y: Hashable, op: Optional[Callable[[Any, Any], Any]] = None*)

Trace shortest path between two existing keys x and y using a breath-first search (refer https://en.wikipedia.org/wiki/Breadth-first_search and https://www.python.org/doc/essays/graphs/).

If *op* is supplied, calculate the combined link / edge value by successively applying operator op to intermediate link values. E.G. To determine xz_val x →[xz_val]→ z we can compute xz_val = op(xy_val, yz_val) if we have x →[xy_val]→ y and y →[yz_val]→ z.

Parameters

- **x,y** (*Hashable*) – Starting and ending node / vertex / key.
- **op** (*Callable[[Any, Any], Any], optional*) – Operator that produces a combined value valid for any two link / edge values, i.e. `result = op(val1, val2)`.

Returns

- **path** (*List*) – List of all points visited along the path including the end nodes, i.e. from x → y: [x, i, j, k, y]. If no path is found it is None.
- **path,value** (*Tuple[List, Any]*) – If *op* is given, the path value is computed and returned along with the path list (as above), if the path covers more than one link. For direct links, the link value is returned. If *op* is given but no path is found this is also None.

Raises KeyError – If *x* or *y* are not verticies, or if *x* == *y*.

pyavia.containers.**wdg_edge**

alias of builtins.slice

2.3 Data Manipulation Modules

2.3.1 Contents

Filtering

pyavia.data.J211_2pole
pyavia.data.J211_4pole

Interpolation

pyavia.data.Interp2Level
pyavia.data.linear_int_ext
pyavia.data.smooth_array2d
pyavia.data.smooth_multi
pyavia.data.subd_num_list

2.3.2 Members

2.4 Fortran

2.4.1 Contents

pyavia.fortran.fortran_do
pyavia.fortran.fortran_array
pyavia.fortran.FortranArray

2.4.2 Members

2.5 Geometry Modules

2.5.1 Contents

Bézier Curves

pyavia.geo.bernstein_poly
pyavia.geo.bezier
pyavia.geo.bezier_deriv

Generic

pyavia.geo.line_pt
pyavia.geo.sine_spacing

2.5.2 Members

2.6 Iterable Modules

2.6.1 Contents

pyavia.iter.all_none	Returns True if all arguments (or elements of a single iterable argument) are None.
pyavia.iter.all_not_none	Shorthand function for `all(x is not None for x in args)`.
pyavia.iter.any_in	Shorthand function for `any(x in target for x in iterable)`.
pyavia.iter.any_none	Returns True if any argument (or element of a single iterable argument) are None.
pyavia.iter.bounded_by	Returns True if *x* is bounded by the given iterable.
pyavia.iter.bracket_list	Returns left and right indicies (*l_idx*, *r_idx*) of a sorted list that bracket *x*.

Continued on next page

Table 11 – continued from previous page

pyavia.iter.count_op	Return a count of the number of items in *it* where `oper(value) == True`.
pyavia.iter.first	Function returning the first item in the iterable that satisfies the condition.
pyavia.iter.flatten	Generator returning entries from a flattened representation of any sequence container (except strings).
pyavia.iter.flatten_list	Generator similar to `flatten`, however only flattens lists until a non-list element is found.
pyavia.iter.split_dict	Split dict *a* by *keys* and return two dicts *x* and *y*:
pyavia.iter.singlify	If *x* only has exactly one element, return that element only.

2.6.2 Members

Handy functions for dealing with various iterables.

pyavia.iter.**all_none**(**args*)
Returns True if all arguments (or elements of a single iterable argument) are None.

pyavia.iter.**all_not_none**(**args*)
Shorthand function for `all(x is not None for x in args)`. Returns True if all **args* are not None, otherwise False.

pyavia.iter.**any_in**(*it: Iterable*, *target*)
Shorthand function for `any(x in target for x in iterable)`. Returns True if found, otherwise False.

pyavia.iter.**any_none**(**args*)
Returns True if any argument (or element of a single iterable argument) are None.

pyavia.iter.**bounded_by**(*x*, *iterable*, *key=None*)
Returns True if *x* is bounded by the given iterable. I.E. `min(iterable) <= x <= max(iterable)`. If *key* function is provided this is applied to *x* and the iterable values before comparison.

pyavia.iter.**bracket_list**(*li*, *x*, *key=None*)
Returns left and right indicies (*l_idx*, *r_idx*) of a sorted list that bracket *x*. I.E. where *li'[*'*l_idx*] <= *x* <= *li'[*'*r_idx*].

Note: This is not the same as the usual one-sided methods which ensure strict inequality on one side (e.g. low <= x < high). This means that for boundary values two brackets may satisfy the condition.

Parameters

- **li** (*List*) – Sorted list / tuple. Sorting can be in either direction.
- **x** – Value to bracket.
- **key** (*function*) – Comparison function if supplied. Default = None.

Returns **l_idx, r_idx** – Note that r_idx = l_idx + 1 on return. For *x* equal to a middle list value, the left side bracket is returned.

Return type Tuple

pyavia.iter.**count_op**(*it: Iterable*, *oper*, *value*)
Return a count of the number of items in *it* where `oper(value) == True`. This allows user-defined objects to be included and is subtly different to `[...].count(...)` which uses the `__eq__` operator.

pyavia.iter.**first**(*it: Iterable*, *condition=<function <lambda>>*)
Function returning the first item in the iterable that satisfies the condition. This function is taken almost directly from: https://stackoverflow.com/a/35513376

```
>>> first((1,2,3), condition=lambda x: x % 2 == 0)
2
>>> first(range(3, 100))
3
>>> first(())
Traceback (most recent call last):
...
StopIteration
```

Parameters

- **it** (*Iterable*) – Iterable to search.
- **condition** (*boolean function (optional)*) – Boolean condition applied to each iterable. If the condition is not given, the first item is returned.

Returns

- *first_item* – First item in the iterable *it* that satisfying the condition.
- *Raises* – StopIteration if no item satysfing the condition is found.

pyavia.iter.**flatten**(*seq*)
Generator returning entries from a flattened representation of any sequence container (except strings). Taken from https://stackoverflow.com/a/2158532

Examples

```
>>> for x in flatten((2, (3, (4, 5), 6), 7)):
...     print(x, end='')
234567
```

Parameters **seq** (*list_like*) – Sequence container

Yields Each entry in turn.

pyavia.iter.**flatten_list**(*li*)
Generator similar to `flatten`, however only flattens lists until a non-list element is found. Note that non-lists may contain sub-lists and these are not flattened.

Parameters **li** (*List*) – List to flatten.

Yields Each non-list entry in turn.

pyavia.iter.**singlify**(*x*)
If *x* only has exactly one element, return that element only. Otherwise return *x* unaltered.

pyavia.iter.**split_dict**(*a: Dict*, *keys: Sequence*) -> *(typing.Dict*, *typing.Dict)*

Split dict *a* by *keys* and return two dicts *x* and *y*:

- *x*: Items in *a* having a key in *keys*.

- *y*: All other items in *a*.

2.7 Math Modules

2.7.1 Contents

pyavia.math.chain_mult	Multiply `start_val` by each element in `li` in turn, producing a List of values the same length of `li`.
pyavia.math.is_number_seq	Returns `True` if `obj` is of `Sequence` type and is not a string / bytes / bytearray.
pyavia.math.kind_atan2	Implementation of atan2 that allows any object as an argument provided it supports __div__ and __float__, allowing units-aware usage.
pyavia.math.kind_div	Tries integer division of x/y before resorting to float.
pyavia.math.min_max	Returns (min, max) of elements in *iterable*.
pyavia.math.monotonic	Returns True if elements of *iterable* are monotonic otherwise false.
pyavia.math.strict_decrease	Shorthand for `monotonic(iterable, -1, strict=True, key=key)`.
pyavia.math.strict_increase	Shorthand for `monotonic(iterable, +1, strict=True, key=key)`.
pyavia.math.vectorise	Applies function `func` to one or more `*values` that can be either scalar or vector.

2.7.2 Members

Useful mathematical functions, particularly if these are not available in NumPy.

`pyavia.math.`**`chain_mult`**(*start_val*, *li: Sequence*) → List
Multiply `start_val` by each element in `li` in turn, producing a List of values the same length of `li`. The starting value is not included.

Examples

```
>>> print(chain_mult(5.0, [2.0, 3.0, 0.5]))
[10.0, 30.0, 15.0]
```

`pyavia.math.`**`is_number_seq`**(*obj*) → bool
Returns `True` if `obj` is of `Sequence` type and is not a string / bytes / bytearray.

`pyavia.math.`**`kind_atan2`**(*y*, *x*) → float
Implementation of atan2 that allows any object as an argument provided it supports __div__ and __float__, allowing units-aware usage.

`pyavia.math.`**`kind_div`**(*x*, *y*) → Union[int, float]
Tries integer division of x/y before resorting to float. If integer division gives no remainder, returns this result otherwise it returns the float result. From https://stackoverflow.com/a/36637240.

`pyavia.math.`**`min_max`**(*iterable*, *key=None*)
Returns (min, max) of elements in *iterable*. Comparison provided using *key* if supplied.

pyavia.math.**monotonic**(*iterable*, *dirn*, *strict=True*, *key=None*)
Returns True if elements of *iterable* are monotonic otherwise false. For *dirn* >= 0 the sequence is strictly increasing i.e. $x_{i+1} > x_i$, otherwise it is strictly decreasing i.e. $x_i + 1 < x_i$. If *strict* = False then equality is sufficient (i.e. >=, <= instead of >, <). Comparison provided using *key* if supplied.

pyavia.math.**strict_decrease**(*iterable*, *key=None*)
Shorthand for `monotonic(iterable, -1, strict=True, key=key)`. Returns True *i.f.f.* all $x_{i+1} < x_i$. Comparison provided using *key* if supplied.

pyavia.math.**strict_increase**(*iterable*, *key=None*)
Shorthand for `monotonic(iterable, +1, strict=True, key=key)`. Returns True *i.f.f.* all $x_{i+1} > x_i$. Comparison provided using *key* if supplied.

pyavia.math.**vectorise**(*func: Callable*, **values*) → Union[list, Any]
Applies function `func` to one or more `*values` that can be either scalar or vector. The function must take the same number of arguments as there are `*values`:

- If all values are numeric sequences, `map()` is used and a list is returned applying `func` to each of the value/s in turn. This applies even if the `len() == 1` for the values.
- If any value is not iterable: A scalar function and result is assumed.

2.8 Propulsion Modules

2.8.1 Contents

Propellers

xxx

Engines - General

xxx

Engines - Gas Turbine

xxx

2.8.2 Members

Functions / objects relating to propulsion analysis such as reciprocating engines, propellers and gas turbines.

2.9 Solver Modules

2.9.1 Contents

`pyavia.solve.bisect_root`	**pyavia.solve** provides functions for finding solutions to various types of equations.
`pyavia.solve.fixed_point`	
`pyavia.solve.newton_bounded`	
`pyavia.solve.solve_dqnm`	
`pyavia.solve.y_to_x`	

2.9.2 Members

2.10 Structures Modules

2.10.1 Contents

pyavia.struct.kt_hole3d
pyavia.struct.mohr2d
pyavia.struct.sn_raithby

2.10.2 Members

2.11 Types Modules

2.11.1 Contents

pyavia.types.coax_type	Try converting *x* into a series of types, returning first result which passes test: next_type(*x*) - x == 0.
pyavia.types.force_type	Try converting *x* into a series of types, with no check on the conversion validity.
pyavia.types.dataclass_fromlist	Initialise a dataclass of type *dc_type* using a list of values *init_vals* ordered to match the class fields (i.e.
pyavia.types.dataclass_names	When passed a type or specific instance of a dataclass, returns an ordered list containing the names of the individual fields.
pyavia.types.make_sentinel	**This factory function is taken directly from ‘‘boltons.typeutils‘‘ and has only cosmetic changes.**

2.11.2 Members

Functions for changing data between different / unusual types.

class pyavia.types.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*)

Bases: typing._Final, typing._Immutable

Type variable.

Usage:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class Generic for more information on generic types. Generic functions work as follows:

def repeat(x: T, n: int) -> List[T]: ‘”Return a list containing n references to x.”’ return [x]*n

def longest(x: A, y: A) -> A: ‘”Return the longest of two strings.”’ return x if len(x) >= len(y) else y

The latter example's signature is essentially the overloading of (str, str) -> str and (bytes, bytes) -> bytes. Also note that if the arguments are instances of some subclass of str, the return type is still plain str.

At runtime, isinstance(x, T) and issubclass(C, T) will raise TypeError.

Type variables defined with covariant=True or contravariant=True can be used to declare covariant or contravariant generic types. See PEP 484 for more details. By default generic types are invariant in all type variables.

Type variables can be introspected. e.g.:

T.__name__ == 'T' T.__constraints__ == () T.__covariant__ == False T.__contravariant__ = False A.__constraints__ == (str, bytes)

Note that only type variables defined in global scope can be pickled.

__bound__

__constraints__

__contravariant__

__covariant__

__init__(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*)
Initialize self. See help(type(self)) for accurate signature.

__name__ = 'TypeVar'

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

pyavia.types.**coax_type**(*x*, **types*, *default=None*)
Try converting *x* into a series of types, returning first result which passes test: next_type(*x*) - x == 0.

Examples

```
>>> coax_type(3.5, int, float)  # float result.
3.5
>>> coax_type(3.0, int, str)  # int result.
3
>>> coax_type("3.0", int, float)  # Error: 3.0 != "3.0".
Traceback (most recent call last):
...
ValueError: Couldn't coax '3.0' to <class 'int'> or <class 'float'>.
>>> xa = 3 + 2j
>>> coax_type(xa, int, float, default=xa)  # Can't conv., gives default.
(3+2j)
```

Parameters

- **x** – Argument to be converted.
- **types** (*list_like*) – Target types to use when trying conversion.
- **default** – Value to return if conversion was unsuccessful.

Returns *x* converted to the first successful type (if possible) or default.

Return type x_converted

Raises `ValueError` – If default is None and conversion was unsuccessful.

`pyavia.types.`**`dataclass_fromlist`**(*dc_type: Type*, *init_vals: Sequence*)
Initialise a dataclass of type *dc_type* using a list of values *init_vals* ordered to match the class fields (i.e. as returned by `dataclass_names`). The length of the *init_vals* cannot exceed the number of dataclass field names. If shorter, remaining fields are unassigned.

`pyavia.types.`**`dataclass_names`**(*dc*) → List[str]
When passed a type or specific instance of a dataclass, returns an ordered list containing the names of the individual fields.

`pyavia.types.`**`fields`**(*class_or_instance*)
Return a tuple describing the fields of this dataclass.

Accepts a dataclass or an instance of one. Tuple elements are of type Field.

`pyavia.types.`**`force_type`**(*x*, **types*)
Try converting *x* into a series of types, with no check on the conversion validity.

Examples

```
>>> force_type(3.5, int, float)
... # Results in an int, because int(x) accepts float and truncates.
3
>>> force_type("3.5+4j", float, complex)
(3.5+4j)
>>> force_type(3.5+4j, int, float, str)
'(3.5+4j)'
```

Parameters

- **`x`** – Argument to be converted.
- **`types`** (*`list_like`*) – Target types to use when trying conversion.

Returns *x* converted to the first successful type, if possible.

Return type x_converted

Raises `ValueError` – If no conversion was possible.

`pyavia.types.`**`is_dataclass`**(*obj*)
Returns True if obj is a dataclass or an instance of a dataclass.

`pyavia.types.`**`make_sentinel`**(*name='_MISSING'*, *var_name=None*)
This factory function is taken directly from ‘‘boltons.typeutils‘‘ and has only cosmetic changes.

Creates and returns a new **instance** of a new class, suitable for usage as a “sentinel”, a kind of singleton often used to indicate a value is missing when `None` is a valid input.

Examples

```
>>> make_sentinel(var_name='_MISSING')
_MISSING
```

Sentinels can be used as default values for optional function arguments, partly because of its less-confusing appearance in automatically generated documentation. Sentinels also function well as placeholders in queues and linked lists.

```
>>> make_sentinel('TEST') == make_sentinel('TEST')
False
>>> type(make_sentinel('TEST')) == type(make_sentinel('TEST'))
False
```

Parameters

- **name** (*str*) – Name of the Sentinel.
- **var_name** (*str (optional)*) – Set this name to the name of the variable in its respective module enable pickleability.

2.12 Units

2.12.1 Contents

pyavia.units.CACHE_COMPUTED_CONVS
pyavia.units.CONV_PATH_LENGTH_WARNING
pyavia.units.OUTPUT_UCODE_PWR
pyavia.units.STD_UNIT_SYSTEM
pyavia.units.Dim
pyavia.units.RealScalar
pyavia.units.RealArray
pyavia.units.DimScalar
pyavia.units.DimArray
pyavia.units.add_base_unit
pyavia.units.add_unit
pyavia.units.convert
pyavia.units.dim
pyavia.units.is_dimarray
pyavia.units.set_conversion
pyavia.units.to_absolute_temp

2.12.2 Members

2.13 Utility Modules

2.13.1 Contents

pyavia.util.temp_filename	Generates a (nearly unique) temporary file name with given *prefix* and *suffix* using a hex UUID, truncated to *rand_length* if required.

2.13.2 Members

Small, general purpose utility functions.

pyavia.util.**temp_filename**(*prefix: str = ''*, *suffix: str = ''*, *rand_length: int = None*)

Generates a (nearly unique) temporary file name with given *prefix* and *suffix* using a hex UUID, truncated to *rand_length* if required. This is useful for interfacing with older DOS and FORTRAN style codes which may have specific rules about filename length.

CHAPTER 3

Copyright

CHAPTER 4

Indices and Tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

F

I

K

M

P

S

T

V

W