src Documentation

Release 0.9.0.5

Pablo Acosta-Serafini

October 29, 2015

Contents

This library provides a collection of utility modules to supplement the Python standard library. The modules provided are:

- **eng**: engineering-related functions including a) handling numbers represented in engineering notation, obtaining their constituent components and converting to and from regular floats; b) pretty printing Numpy vectors; and c) formatting numbers represented in scientific notation with a greater degree of control and options than standard Python string formatting
- **exdoc**: automatically generate exceptions documentation marked up in reStructuredText with help from cog and the exh module
- **exh**: register exceptions and then raise them if a given condition is true
- **misc**: miscellaneous utility functions that can be applied in a variety of circumstances; there are context managers, membership functions (test if an argument is of a given type), numerical functions and string functions
- **pcontracts**: thin wrapper around the PyContracts library that enables customization of the exception type raised and limited customization of the exception message
- **pcsv**: handle comma-separated values (CSV) files and do lightweight processing of their data
- **pinspect**: supplements Python's introspection capabilities
- **plot**: create high-quality, presentation-ready X-Y graphs quickly and easily
- **ptypes**: several pseudo-type definitions which can be enforced and/or validated with custom contracts defined using the pcontracts module
- **test**: functions to aid in the unit testing of modules in the package (py.test-based)
- **tree**: build, handle, process and search tries

CHAPTER 1

Interpreter

The package has been developed and tested with Python 2.7 and Python 3.4

CHAPTER 2

Installing

```
$ pip install putil
```

CHAPTER 3

Documentation

Available at Read the Docs

CHAPTER 4

Contributing

1. The repository may be forked from Bitbucket; clone the forked repository recursively since the Read the Docs documentation theme is a repository sub-module [1]:

```
$ git clone --recursive \
      https://bitbucket.org/[bitbucket-user-name]/putil.git
$ cd putil
$ export PUTIL_DIR=${PWD}
```

2. Install the project's Git hooks. The pre-commit hook does some minor consistency checks, namely trailing whitespace and PEP8 compliance via Pylint. Assuming the directory to which the repository was cloned is in the `$PUTIL_DIR` shell environment variable:

```
$ ${PUTIL_DIR}/sbin/setup-git-hooks.sh
```

3. Ensure that the Python interpreter can find the package modules (update the `$PYTHONPATH` environment variable, or use sys.paths(), etc.)

```
$ export PYTHONPATH=${PYTHONPATH}:${PUTIL_DIR}
```

   This is relevant only if it is desired to run unit tests, measure test coverage and/or (re)build the documentation using the cloned repository (and not a virtual environment). This option is attractive as it allows for faster iterations, but final pre-commit validation should be done using the tox flow (`pkg-validate.sh` script, see below)

4. Install the dependencies (if needed):

   - Cog >= 2.4
   - Coverage >= 3.7.1
   - Decorator >= 3.4.2
   - Funcsigs >= 0.4 (only for Python 2.7)
   - Matplotlib >= 1.3.1
   - Mock >= 1.0.1 (only for Python 2.7)
   - Numpy >= 1.8.2
   - Pillow >= 2.6.1
   - PyContracts >= 1.7.2
   - Pytest-coverage >= 1.8.0

[1] All examples are for the bash shell

- Pytest-xdist >= 1.8.0 (optional)
- Py.test >= 2.7.0
- Scipy >= 0.13.3
- Six >= 1.4.0
- Sphinx >= 1.2.3
- Tox >= 1.9.0

5. Write a unit test which shows that a bug was fixed or that a new feature or API works as expected. Run the package tests to ensure that the bug fix or new feature does not have adverse side effects. If possible achieve 100% code and branch coverage of the contribution. Thorough package validation can be done via setuptools:

```
$ python setup.py tests
running tests
running egg_info
writing requirements to putil.egg-info/requires.txt
writing putil.egg-info/PKG-INFO
...
```

Setuptools runs tox with its two default environments `py27-pkg` and `py34-pkg`. These use the Python 2.7 and 3.4 interpreters to test all code in the documentation (both in Sphinx `*.rst` source files and in docstrings), run all unit tests and re-build the exceptions documentation. To pass arguments to tox use the `-a` option followed by a quoted string. For example:

```
$ python setup.py tests -a "-e py27-pkg -- -n 4"
running tests
...
```

There are other convenience environments defined for tox [2]:

- `py27-repl` and `py34-repl` run the Python 2.7 or Python 3.4 interpreter in the appropriate virtual environment. The `putil` package is pip-installed by tox when the environments are created
- `py27-test` and `py34-test` run py.test using the Python 2.7 or Python 3.4 interpreter in the appropriate virtual environment. Arguments to py.test can be passed in the command line after a double dash (`--`) , for example:

```
$ tox -e py34-test -- -x test_eng.py
GLOB sdist-make: [...]/putil/setup.py
py34-test inst-nodeps: [...]/putil/.tox/dist/putil-[...].zip
py34-test runtests: PYTHONHASHSEED='680528711'
py34-test runtests: commands[0] | [...]py.test -x test_eng.py
==================== test session starts ====================
platform linux -- Python 3.4.2 -- py-1.4.30 -- [...]
...
```

- `py27-cov` and `py34-cov` test code and branch coverage using the Python 2.7 or Python 3.4 interpreter in the appropriate virtual environment. Arguments to py.test can be passed in the command line after a double dash (`--`). The report can be found in `${PUTIL_DIR}/.tox/py27/usr/share/putil/tests/htmlcov/index.html` or `${PUTIL_DIR}/.tox/py34/usr/share/putil/tests/htmlcovindex.html` depending on the interpreter used.

6. The `${PUTIL_DIR}/sbin` directory contains other relevant development scripts:

- **build-docs.sh:** (re)builds the package documentation

[2] Tox configuration largely inspired by Ionel's codelog

```
$ ${PUTIL_DIR}/sbin/build-docs.sh -h
build-docs.sh

Usage:
  build-docs.sh -h
  build-docs.sh -r -t [-d dir] [-n num-cpus] [module-name]
  build-docs.sh [-d dir] [module-name]

Options:
  -h  Show this screen
  -r  Rebuild exceptions documentation. If no module name
      is given all modules with auto-generated exceptions
      documentation are rebuilt
  -d  Specify source file directory
      [default: (build-docs.sh directory)/../putil]
  -t  Diff original and rebuilt file(s) (exit code 0
      indicates file(s) are identical, exit code 1
      indicates file(s) are different
  -n  Number of CPUs to use [default: 1]
```

- **build-tags.sh:** builds the project's exuberant ctags file `${PUTIL_DIR}/tags`

```
$ ${PUTIL_DIR}/sbin/build-tags.sh -h
build-tags.sh

Usage:
  build-tags.sh -h
  build-tags.sh

Options:
  -h  Show this screen
```

- **gen_ref_images.py:** (re)generates the plot module reference images needed for unit testing

```
$ ${PUTIL_DIR}/sbin/gen_ref_images.py
Generating image [PUTIL_DIR]/tests/support/...
...
```

7. Contributors must follow the PyPA Code of Conduct

CHAPTER 5

License

CHAPTER 6

Contents

6.1 Putil Library

This library provides a collection of utility modules to supplement the Python standard library. The modules provided are:

- **eng**: engineering-related functions including a) handling numbers represented in engineering notation, obtaining their constituent components and converting to and from regular floats; b) pretty printing Numpy vectors; and c) formatting numbers represented in scientific notation with a greater degree of control and options than standard Python string formatting
- **exdoc**: automatically generate exceptions documentation marked up in reStructuredText with help from cog and the exh module
- **exh**: register exceptions and then raise them if a given condition is true
- **misc**: miscellaneous utility functions that can be applied in a variety of circumstances; there are context managers, membership functions (test if an argument is of a given type), numerical functions and string functions
- **pcontracts**: thin wrapper around the PyContracts library that enables customization of the exception type raised and limited customization of the exception message
- **pcsv**: handle comma-separated values (CSV) files and do lightweight processing of their data
- **pinspect**: supplements Python's introspection capabilities
- **plot**: create high-quality, presentation-ready X-Y graphs quickly and easily
- **ptypes**: several pseudo-type definitions which can be enforced and/or validated with custom contracts defined using the pcontracts module
- **test**: functions to aid in the unit testing of modules in the package (py.test-based)
- **tree**: build, handle, process and search tries

6.2 Interpreter

The package has been developed and tested with Python 2.7 and Python 3.4

6.3 Installing

```
$ pip install putil
```

6.4 Documentation

Available at Read the Docs

6.5 Contributing

1. The repository may be forked from Bitbucket; clone the forked repository recursively since the Read the Docs documentation theme is a repository sub-module [1]:

```
$ git clone --recursive \
      https://bitbucket.org/[bitbucket-user-name]/putil.git
$ cd putil
$ export PUTIL_DIR=${PWD}
```

2. Install the project's Git hooks. The pre-commit hook does some minor consistency checks, namely trailing whitespace and PEP8 compliance via Pylint. Assuming the directory to which the repository was cloned is in the `$PUTIL_DIR` shell environment variable:

```
$ ${PUTIL_DIR}/sbin/setup-git-hooks.sh
```

3. Ensure that the Python interpreter can find the package modules (update the `$PYTHONPATH` environment variable, or use sys.paths(), etc.)

```
$ export PYTHONPATH=${PYTHONPATH}:${PUTIL_DIR}
```

   This is relevant only if it is desired to run unit tests, measure test coverage and/or (re)build the documentation using the cloned repository (and not a virtual environment). This option is attractive as it allows for faster iterations, but final pre-commit validation should be done using the tox flow (`pkg-validate.sh` script, see below)

4. Install the dependencies (if needed):

   - Cog >= 2.4
   - Coverage >= 3.7.1
   - Decorator >= 3.4.2
   - Funcsigs >= 0.4 (only for Python 2.7)
   - Matplotlib >= 1.3.1
   - Mock >= 1.0.1 (only for Python 2.7)
   - Numpy >= 1.8.2
   - Pillow >= 2.6.1
   - PyContracts >= 1.7.2
   - Pytest-coverage >= 1.8.0

[1] All examples are for the bash shell

- Pytest-xdist >= 1.8.0 (optional)
- Py.test >= 2.7.0
- Scipy >= 0.13.3
- Six >= 1.4.0
- Sphinx >= 1.2.3
- Tox >= 1.9.0

5. Write a unit test which shows that a bug was fixed or that a new feature or API works as expected. Run the package tests to ensure that the bug fix or new feature does not have adverse side effects. If possible achieve 100% code and branch coverage of the contribution. Thorough package validation can be done via setuptools:

```
$ python setup.py tests
running tests
running egg_info
writing requirements to putil.egg-info/requires.txt
writing putil.egg-info/PKG-INFO
...
```

Setuptools runs tox with its two default environments `py27-pkg` and `py34-pkg`. These use the Python 2.7 and 3.4 interpreters to test all code in the documentation (both in Sphinx `*.rst` source files and in docstrings), run all unit tests and re-build the exceptions documentation. To pass arguments to tox use the `-a` option followed by a quoted string. For example:

```
$ python setup.py tests -a "-e py27-pkg -- -n 4"
running tests
...
```

There are other convenience environments defined for tox [2]:

- `py27-repl` and `py34-repl` run the Python 2.7 or Python 3.4 interpreter in the appropriate virtual environment. The `putil` package is pip-installed by tox when the environments are created
- `py27-test` and `py34-test` run py.test using the Python 2.7 or Python 3.4 interpreter in the appropriate virtual environment. Arguments to py.test can be passed in the command line after a double dash (`--`) , for example:

```
$ tox -e py34-test -- -x test_eng.py
GLOB sdist-make: [...]/putil/setup.py
py34-test inst-nodeps: [...]/putil/.tox/dist/putil-[...].zip
py34-test runtests: PYTHONHASHSEED='680528711'
py34-test runtests: commands[0] | [...]py.test -x test_eng.py
==================== test session starts ====================
platform linux -- Python 3.4.2 -- py-1.4.30 -- [...]
...
```

- `py27-cov` and `py34-cov` test code and branch coverage using the Python 2.7 or Python 3.4 interpreter in the appropriate virtual environment. Arguments to py.test can be passed in the command line after a double dash (`--`). The report can be found in `${PUTIL_DIR}/.tox/py27/usr/share/putil/tests/htmlcov/index.html` or `${PUTIL_DIR}/.tox/py34/usr/share/putil/tests/htmlcovindex.html` depending on the interpreter used.

6. The `${PUTIL_DIR}/sbin` directory contains other relevant development scripts:

- **build-docs.sh:** (re)builds the package documentation

[2] Tox configuration largely inspired by Ionel's codelog

```
$ ${PUTIL_DIR}/sbin/build-docs.sh -h
build-docs.sh

Usage:
  build-docs.sh -h
  build-docs.sh -r -t [-d dir] [-n num-cpus] [module-name]
  build-docs.sh [-d dir] [module-name]

Options:
  -h  Show this screen
  -r  Rebuild exceptions documentation. If no module name
      is given all modules with auto-generated exceptions
      documentation are rebuilt
  -d  Specify source file directory
      [default: (build-docs.sh directory)/../putil]
  -t  Diff original and rebuilt file(s) (exit code 0
      indicates file(s) are identical, exit code 1
      indicates file(s) are different
  -n  Number of CPUs to use [default: 1]
```

- **build-tags.sh:** builds the project's exuberant ctags file `${PUTIL_DIR}/tags`

```
$ ${PUTIL_DIR}/sbin/build-tags.sh -h
build-tags.sh

Usage:
  build-tags.sh -h
  build-tags.sh

Options:
  -h  Show this screen
```

- **gen_ref_images.py:** (re)generates the plot module reference images needed for unit testing

```
$ ${PUTIL_DIR}/sbin/gen_ref_images.py
Generating image [PUTIL_DIR]/tests/support/...
...
```

7. Contributors must follow the PyPA Code of Conduct

6.6 License

The MIT License (MIT)

Copyright (c) 2013-2015 Pablo Acosta-Serafini

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PAR-

TICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

6.7 eng module

This module provides engineering-related functions including:

- Handling numbers represented in engineering notation, obtaining their constituent components and converting to and from regular floats. For example:

```
>>> import putil.eng
>>> x = putil.eng.peng(1346, 2, True)
>>> x
'   1.35k'
>>> putil.eng.peng_float(x)
1350.0
>>> putil.eng.peng_int(x)
1
>>> putil.eng.peng_frac(x)
35
>>> str(putil.eng.peng_mant(x))
'1.35'
>>> putil.eng.peng_power(x)
EngPower(suffix='k', exp=1000.0)
>>> putil.eng.peng_suffix(x)
'k'
```

- Pretty printing Numpy vectors. For example:

```
>>> from __future__ import print_function
>>> import putil.eng
>>> header = 'Vector: '
>>> data = [1e-3, 20e-6, 30e+6, 4e-12, 5.25e3, -6e-9, 70, 8, 9]
>>> print(
...     header+putil.eng.pprint_vector(
...         data,
...         width=30,
...         eng=True,
...         frac_length=1,
...         limit=True,
...         indent=len(header)
...     )
... )
Vector: [    1.0m,   20.0u,   30.0M,
                     ...
             70.0 ,    8.0 ,    9.0  ]
```

- Formatting numbers represented in scientific notation with a greater degree of control and options than standard Python string formatting. For example:

```
>>> import putil.eng
>>> putil.eng.to_scientific_string(
...     number=99.999,
...     frac_length=1,
...     exp_length=2,
```

```
...     sign_always=True
... )
'+1.0E+02'
```

6.7.1 Named tuples

putil.eng.**ENGPOWER**(*suffix*, *exp*)
Constructor for engineering notation suffix

putil.eng.**NUMCOMP**(*mant*, *exp*)
Constructor for number components representation

6.7.2 Functions

putil.eng.**no_exp**(*number*)
Converts a number to a string guaranteeing that the result is not expressed in scientific notation

Parameters **number** (*integer or float*) – Number to convert

Return type string

Raises RuntimeError (Argument ‘number‘ is not valid)

putil.eng.**peng**(*number*, *frac_length*, *rjust=True*)
Converts a number to engineering notation. The absolute value of the number (if it is not exactly zero) is bounded to the interval [1E-24, 1E+24)

Parameters

- **number** (*integer or float*) – Number to convert
- **frac_length** (*NonNegativeInteger*) – Number of digits of fractional part
- **rjust** (*boolean*) – Flag that indicates whether the number is right-justified (True) or not (False)

Return type string

Raises

- RuntimeError (Argument ‘frac_length‘ is not valid)
- RuntimeError (Argument ‘number‘ is not valid)
- RuntimeError (Argument ‘rjust‘ is not valid)

The supported engineering suffixes are:

Exponent	Name	Suffix
1E-24	yocto	y
1E-21	zepto	z
1E-18	atto	a
1E-15	femto	f
1E-12	pico	p
1E-9	nano	n
1E-6	micro	u
1E-3	milli	m
1E+0		
1E+3	kilo	k
1E+6	mega	M
1E+9	giga	G
1E+12	tera	T
1E+15	peta	P
1E+18	exa	E
1E+21	zetta	Z
1E+24	yotta	Y

For example:

```
>>> import putil.eng
>>> putil.eng.peng(1235.6789E3, 3, False)
'1.236M'
```

putil.eng.**peng_float**(*snum*)

Returns the floating point equivalent of a number represented in engineering notation

Parameters **snum** (*EngineeringNotationNumber*) – Number

Return type string

Raises RuntimeError (Argument ‘snum‘ is not valid)

For example:

```
>>> import putil.eng
>>> putil.eng.peng_float(putil.eng.peng(1235.6789E3, 3, False))
1236000.0
```

putil.eng.**peng_frac**(*snum*)

Returns the fractional part of a number represented in engineering notation

Parameters **snum** (*EngineeringNotationNumber*) – Number

Return type integer

Raises RuntimeError (Argument ‘snum‘ is not valid)

For example:

```
>>> import putil.eng
>>> putil.eng.peng_frac(putil.eng.peng(1235.6789E3, 3, False))
236
```

putil.eng.**peng_int**(*snum*)

Returns the integer part of a number represented in engineering notation

Parameters **snum** (*EngineeringNotationNumber*) – Number

Return type integer

Raises RuntimeError (Argument ‘snum‘ is not valid)

For example:

```
>>> import putil.eng
>>> putil.eng.peng_int(putil.eng.peng(1235.6789E3, 3, False))
1
```

putil.eng.**peng_mant**(*snum*)

Returns the mantissa of a number represented in engineering notation

Parameters **snum** (*EngineeringNotationNumber*) – Number

Return type float

Raises RuntimeError (Argument ‘snum‘ is not valid)

For example:

```
>>> import putil.eng
>>> putil.eng.peng_mant(putil.eng.peng(1235.6789E3, 3, False))
1.236
```

putil.eng.**peng_power**(*snum*)

Returns engineering suffix and floating point equivalent of the suffix when a number is represented in engineering notation. *putil.eng.peng()* lists the correspondence between suffix and floating point exponent.

Parameters **snum** (*EngineeringNotationNumber*) – Number

Return type named tuple in which the first item is the engineering suffix and the second item is the floating point equivalent of the suffix when the number is represented in engineering notation.

Raises RuntimeError (Argument ‘snum‘ is not valid)

For example:

```
>>> import putil.eng
>>> putil.eng.peng_power(putil.eng.peng(1235.6789E3, 3, False))
EngPower(suffix='M', exp=1000000.0)
```

putil.eng.**peng_suffix**(*snum*)

Returns the suffix of a number represented in engineering notation

Parameters **snum** (*EngineeringNotationNumber*) – Number

Return type string

Raises RuntimeError (Argument ‘snum‘ is not valid)

For example:

```
>>> import putil.eng
>>> putil.eng.peng_suffix(putil.eng.peng(1235.6789E3, 3, False))
'M'
```

putil.eng.**peng_suffix_math**(*suffix*, *offset*)

Returns an engineering suffix based on a starting suffix and an offset of number of suffixes

Parameters

- **suffix** (*EngineeringNotationSuffix*) – Engineering suffix
- **offset** (*integer*) – Engineering suffix offset

Return type string

Raises

- RuntimeError (Argument ‘offset‘ is not valid)
- RuntimeError (Argument ‘suffix‘ is not valid)
- ValueError (Argument ‘offset‘ is not valid)

For example:

```
>>> import putil.eng
>>> putil.eng.peng_suffix_math('u', 6)
'T'
```

putil.eng.**pprint_vector**(*vector*, *limit=False*, *width=None*, *indent=0*, *eng=False*, *frac_length=3*)

Formats a list of numbers (vector) or a Numpy vector for printing. If the argument **vector** is `None` the string `'None'` is returned

Parameters

- **vector** (*list of integers or floats, Numpy vector or None*) – Vector to pretty print or None
- **limit** (*boolean*) – Flag that indicates whether at most 6 vector items are printed (all vector items if its length is equal or less than 6, first and last 3 vector items if it is not) (True), or the entire vector is printed (False)
- **width** (*integer or None*) – Number of available characters per line. If None the vector is printed in one line
- **indent** (*boolean*) – Flag that indicates whether all subsequent lines after the first one are indented (True) or not (False). Only relevant if **width** is not None
- **eng** (*boolean*) – Flag that indicates whether engineering notation is used (True) or not (False)
- **frac_length** (*integer*) – Number of digits of fractional part (only applicable if **eng** is True)

Raises ValueError (Argument ‘width‘ is too small)

Return type string

For example:

```
>>> from __future__ import print_function
>>> import putil.eng
>>> header = 'Vector: '
>>> data = [1e-3, 20e-6, 300e+6, 4e-12, 5.25e3, -6e-9, 700, 8, 9]
>>> print(
...     header+putil.eng.pprint_vector(
...         data,
...         width=30,
...         eng=True,
...         frac_length=1,
...         limit=True,
...         indent=len(header)
...     )
... )
```

```
    Vector: [    1.0m,   20.0u,  300.0M,
                         ...
              700.0 ,    8.0 ,    9.0  ]
    >>> print(
    ...     header+putil.eng.pprint_vector(
    ...         data,
    ...         width=30,
    ...         eng=True,
    ...         frac_length=0,
    ...         indent=len(header)
    ...     )
    ... )
    Vector: [    1m,   20u,  300M,    4p,
                 5k,   -6n,  700 ,    8 ,
                 9  ]
    >>> print(putil.eng.pprint_vector(data, eng=True, frac_length=0))
    [    1m,   20u,  300M,    4p,    5k,   -6n,  700 ,    8 ,    9  ]
    >>> print(putil.eng.pprint_vector(data, limit=True))
    [ 0.001, 2e-05, 300000000.0, ..., 700, 8, 9 ]
```

putil.eng.**round_mantissa**(*arg*, *decimals=0*)

Rounds the fractional part of a floating point number mantissa or Numpy vector of floating point numbers to a given number of digits. Integers are not altered. The mantissa used is that of the floating point number(s) when expressed in normalized scientific notation

Parameters

- **arg** (*integer, float, Numpy vector of integers or floats, or None*) – Input data
- **decimals** (*integer*) – Number of digits to round the fractional part of the mantissa to.

Return type same as **arg**

For example:

```
    >>> import putil.eng
    >>> putil.eng.round_mantissa(012345678E-6, 3)
    12.35
    >>> putil.eng.round_mantissa(5, 3)
    5
```

putil.eng.**to_scientific_string**(*number*, *frac_length=None*, *exp_length=None*, *sign_always=False*)

Converts a number or a string representing a number to a string with the number expressed in scientific notation. Full precision is maintained if the number is represented as a string

Parameters

- **number** (*number or string*) – Number to convert
- **frac_length** (*integer or None*) – Number of digits of fractional part, None indicates that the fractional part of the number should not be limited
- **exp_length** (*integer or None*) – Number of digits of the exponent; the actual length of the exponent takes precedence if it is longer
- **sign_always** (*boolean*) – Flag that indicates whether the sign always precedes the number for both non-negative and negative numbers (True) or only for negative numbers (False)

Return type string

For example:

```
>>> import putil.eng
>>> putil.eng.to_scientific_string(333)
'3.33E+2'
>>> putil.eng.to_scientific_string(0.00101)
'1.01E-3'
>>> putil.eng.to_scientific_string(99.999, 1, 2, True)
'+1.0E+02'
```

putil.eng.**to_scientific_tuple**(*number*)

Returns mantissa and exponent of a number when expressed in scientific notation. Full precision is maintained if the number is represented as a string

Parameters **number** (*integer, float or string*) – Number

Return type named tuple in which the first item is the mantissa (*string*) and the second item is the exponent (*integer*) of the number when expressed in scientific notation

For example:

```
>>> import putil.eng
>>> putil.eng.to_scientific_tuple('135.56E-8')
NumComp(mant='1.3556', exp=-6)
>>> putil.eng.to_scientific_tuple(0.0000013556)
NumComp(mant='1.3556', exp=-6)
```

6.8 exdoc module

This module can be used to automatically generate exceptions documentation marked up in reStructuredText with help from cog and the *exh module*.

The exceptions to auto-document need to be "traced" before the documentation is generated; in general tracing consists of calling the methods, functions and/or class properties so that all the required *putil.exh.ExHandle.add_exception()* calls are covered (exceptions generated by contracts defined using the *pcontracts module* are automatically traced when the contracts are checked). A convenient way of tracing a module is to simply run its test suite, provided that it covers the exceptions that need to be documented.

For example, it is desired to auto-document the exceptions of a module `my_module.py`, which has tests in `test_my_module.py`. Then a tracing module `trace_my_module.py` can be created to leverage the already written tests:

```
# trace_my_module_1.py
# Option 1: use already written test bench
from __future__ import print_function
import copy, os, pytest, putil.exdoc

def trace_module(no_print=True):
    """ Trace my_module exceptions """
    pwd = os.path.dirname(__file__)
    script_name = os.path.join(pwd, 'test_my_module.py')
    with putil.exdoc.ExDocCxt() as exdoc_obj:
        if pytest.main('-s -vv -x {0}'.format(script_name)):
            raise RuntimeError(
                'Tracing did not complete successfully'
            )
    if not no_print:
        module_prefix = 'docs.support.my_module.'
        callable_names = ['func', 'MyClass.value']
```

```
        for callable_name in callable_names:
            callable_name = module_prefix+callable_name
            print('\nCallable: {0}'.format(callable_name))
            print(exdoc_obj.get_sphinx_doc(callable_name, width=70))
            print('\n')
    return copy.copy(exdoc_obj)

if __name__ == '__main__':
    trace_module(False)
```

The context manager *putil.exdoc.ExDocCxt* sets up the tracing environment and returns a *putil.exdoc.ExDoc* object that can the be used in the documentation string of each callable to extract the exceptions documentation. In this example it is assumed that the tests are written using pytest, but any test framework can be used. Another way to trace the module is to simply call all the functions, methods or class properties that need to be documented. For example:

```
# trace_my_module_2.py
# Option 2: manually use all callables to document
from __future__ import print_function
import copy, putil.exdoc, docs.support.my_module

def trace_module(no_print=True):
    """ Trace my_module_original exceptions """
    with putil.exdoc.ExDocCxt() as exdoc_obj:
        try:
            docs.support.my_module.func('John')
            obj = docs.support.my_module.MyClass()
            obj.value = 5
            obj.value
        except:
            raise RuntimeError(
                'Tracing did not complete successfully'
            )
    if not no_print:
        module_prefix = 'docs.support.my_module.'
        callable_names = ['func', 'MyClass.value']
        for callable_name in callable_names:
            callable_name = module_prefix+callable_name
            print('\nCallable: {0}'.format(callable_name))
            print(exdoc_obj.get_sphinx_doc(callable_name, width=70))
            print('\n')
    return copy.copy(exdoc_obj)

if __name__ == '__main__':
    trace_module(False)
```

And the actual module `my_module` code is (before auto-documentation):

```
# my_module.py
# Exception tracing initialization code
"""
[[[cog
import os, sys
sys.path.append(os.environ['TRACER_DIR'])
import trace_my_module_1
exobj = trace_my_module_1.trace_module(no_print=True)
]]]
[[[end]]]
```

```
"""

import putil.exh

def func(name):
    r"""
    Prints your name

    :param   name: Name to print
    :type name: string

    .. [[[cog cog.out(exobj.get_sphinx_autodoc(width=69))]]]
    .. [[[end]]]

    """
    exhobj = putil.exh.get_or_create_exh_obj()
    exhobj.add_exception(
        exname='illegal_name',
        extype=TypeError,
        exmsg='Argument `name` is not valid'
    )
    exhobj.raise_exception_if(
        exname='illegal_name',
        condition=not isinstance(name, str)
    )
    return 'My name is {0}'.format(name)

class MyClass(object):
    """
    Stores a value

    :param  value: value
    :type   value: integer

    .. [[[cog cog.out(exobj.get_sphinx_autodoc(width=69))]]]
    .. [[[end]]]
    """
    def __init__(self, value=None):
        self._exhobj = putil.exh.get_or_create_exh_obj()
        self._value = None if not value else value

    def _get_value(self):
        self._exhobj.add_exception(
            exname='not_set',
            extype=RuntimeError,
            exmsg='Attribute `value` not set'
        )
        self._exhobj.raise_exception_if(
            exname='not_set',
            condition=not self._value
        )
        return self._value

    def _set_value(self, value):
        self._exhobj.add_exception(
            exname='illegal',
            extype=RuntimeError,
            exmsg='Argument `value` is not valid'
```

```
        )
        self._exhobj.raise_exception_if(
            exname='illegal',
            condition=not isinstance(value, int)
        )
        self._value = value

    value = property(_get_value, _set_value)
    r"""
    Sets or returns a value

    :type:  integer
    :rtype: integer or None

    .. [[[cog cog.out(exobj.get_sphinx_autodoc(width=69))]]]
    .. [[[end]]]
    """
```

A simple shell script can be written to automate the cogging of the `my_module.py` file:

```
#!/bin/bash
set -e

finish() {
    export TRACER_DIR=""
    cd ${cpwd}
}
trap finish EXIT

input_file=${1:-my_module.py}
output_file=${2:-my_module.py}
export TRACER_DIR=$(dirname ${input_file})
cog.py -e -x -o ${input_file}.tmp ${input_file} > /dev/null && \
    mv -f ${input_file}.tmp ${input_file}
cog.py -e -o ${input_file}.tmp ${input_file} > /dev/null && \
    mv -f ${input_file}.tmp ${output_file}
```

After the script is run and the auto-documentation generated, each callable has a reStructuredText marked-up `:raises:` section:

```
# my_module_ref.py
# Exception tracing initialization code
"""
[[[cog
import os, sys
sys.path.append(os.environ['TRACER_DIR'])
import trace_my_module_1
exobj = trace_my_module_1.trace_module(no_print=True)
]]]
[[[end]]]
"""

import putil.exh

def func(name):
    r"""
    Prints your name

    :param   name: Name to print
```

```
    :type name: string

    .. [[[cog cog.out(exobj.get_sphinx_autodoc(width=69))]]]
    .. Auto-generated exceptions documentation for
    .. docs.support.my_module.func

    :raises: TypeError (Argument \`name\` is not valid)

    .. [[[end]]]

    """
    exhobj = putil.exh.get_or_create_exh_obj()
    exhobj.add_exception(
        exname='illegal_name',
        extype=TypeError,
        exmsg='Argument `name` is not valid'
    )
    exhobj.raise_exception_if(
        exname='illegal_name',
        condition=not isinstance(name, str)
    )
    return 'My name is {0}'.format(name)

class MyClass(object):
    """
    Stores a value

    :param  value: value
    :type   value: integer

    .. [[[cog cog.out(exobj.get_sphinx_autodoc(width=69))]]]
    .. [[[end]]]
    """
    def __init__(self, value=None):
        self._exhobj = putil.exh.get_or_create_exh_obj()
        self._value = None if not value else value

    def _get_value(self):
        self._exhobj.add_exception(
            exname='not_set',
            extype=RuntimeError,
            exmsg='Attribute `value` not set'
        )
        self._exhobj.raise_exception_if(
            exname='not_set',
            condition=not self._value
        )
        return self._value

    def _set_value(self, value):
        self._exhobj.add_exception(
            exname='illegal',
            extype=RuntimeError,
            exmsg='Argument `value` is not valid'
        )
        self._exhobj.raise_exception_if(
            exname='illegal',
            condition=not isinstance(value, int)
```

```
        )
        self._value = value

    value = property(_get_value, _set_value)
    r"""
    Sets or returns a value

    :type:  integer
    :rtype: integer or None

    .. [[[cog cog.out(exobj.get_sphinx_autodoc(width=69))]]]
    .. Auto-generated exceptions documentation for
    .. docs.support.my_module.MyClass.value

    :raises:
     * When assigned

       * RuntimeError (Argument \`value\` is not valid)

     * When retrieved

       * RuntimeError (Attribute \`value\` not set)

    .. [[[end]]]
    """
```

Warning: Due to the limited introspection capabilities of class properties, only properties defined using the property built-in function can be documented with *putil.exdoc.ExDoc.get_sphinx_autodoc()*. Properties defined by other methods can still be auto-documented with *putil.exdoc.ExDoc.get_sphinx_doc()* and explicitly providing the method/function name.

6.8.1 Classes

class putil.exdoc.**ExDocCxt**(*exclude=None*, *pickle_fname=None*, *in_callables_fname=None*, *out_callables_fname=None*)

Bases: object

Context manager to simplify exception tracing; it sets up the tracing environment and returns a *putil.exdoc.ExDoc* object that can the be used in the documentation string of each callable to extract the exceptions documentation with either *putil.exdoc.ExDoc.get_sphinx_doc()* or *putil.exdoc.ExDoc.get_sphinx_autodoc()*.

Parameters

- **exclude** (*list of strings or None*) – Module exclusion list. A particular callable in an otherwise fully qualified name is omitted if it belongs to a module in this list. If None all callables are included
- **pickle_fname** (*FileName* or None) – File name to pickle traced exception handler (useful for debugging purposes). If None all pickle file is created
- **in_callables_fname** (*FileNameExists* or None) – File name that contains traced modules information. File can be produced by either the *putil.pinspect.Callables.save()* or *putil.exh.ExHandle.save_callables()* methods

- **out_callables_fname** (*FileNameExists* or None) – File name to save traced modules information to in JSON format. If the file exists it is overwritten

Raises

- OSError (File *[in_callables_fname]* could not be found)
- RuntimeError (Argument 'in_callables_fname' is not valid)
- RuntimeError (Argument 'exclude' is not valid)
- RuntimeError (Argument 'out_callables_fname' is not valid)
- RuntimeError (Argument 'pickle_fname' is not valid)

For example:

```
>>> from __future__ import print_function
>>> import putil.eng, putil.exdoc
>>> with putil.exdoc.ExDocCxt() as exdoc_obj:
...     value = putil.eng.peng(1e6, 3, False)
>>> print(exdoc_obj.get_sphinx_doc('putil.eng.peng'))
.. Auto-generated exceptions documentation for putil.eng.peng

:raises:
 * RuntimeError (Argument \`frac_length\` is not valid)

 * RuntimeError (Argument \`number\` is not valid)

 * RuntimeError (Argument \`rjust\` is not valid)
```

class putil.exdoc.**ExDoc**(*exh_obj*, *depth=None*, *exclude=None*)

Bases: object

Generates exception documentation with reStructuredText mark-up

Parameters

- **exh_obj** (*putil.exh.ExHandle*) – Exception handler containing exception information for the callable(s) to be documented
- **depth** (*non-negative integer or None*) – Default hierarchy levels to include in the exceptions per callable (see *putil.exdoc.ExDoc.depth*). If None exceptions at all depths are included
- **exclude** (*list of strings or None*) – Default list of (potentially partial) module and callable names to exclude from exceptions per callable (see *putil.exdoc.ExDoc.exclude*). If None all callables are included

Return type *putil.exdoc.ExDoc*

Raises

- RuntimeError (Argument 'depth' is not valid)
- RuntimeError (Argument 'exclude' is not valid)
- RuntimeError (Argument 'exh_obj' is not valid)
- RuntimeError (Exceptions database is empty)
- RuntimeError (Exceptions do not have a common callable)
- ValueError (Object of argument 'exh_obj' does not have any exception trace information)

get_sphinx_autodoc(*depth=None*, *exclude=None*, *width=72*, *error=False*, *raised=False*)
Returns an exception list marked up in reStructuredText automatically determining callable name

Parameters

- **depth** (*non-negative integer or None*) – Hierarchy levels to include in the exceptions list (overrides default **depth** argument; see `putil.exdoc.ExDoc.depth`). If None exceptions at all depths are included
- **exclude** (*list of strings or None*) – List of (potentially partial) module and callable names to exclude from exceptions list (overrides default **exclude** argument, see `putil.exdoc.ExDoc.exclude`). If None all callables are included
- **width** (*integer*) – Maximum width of the lines of text (minimum 40)
- **error** (*boolean*) – Flag that indicates whether an exception should be raised if the callable is not found in the callables exceptions database (True) or not (False)
- **raised** (*boolean*) – Flag that indicates whether only exceptions that were raised (and presumably caught) should be documented (True) or all registered exceptions should be documented (False)

Raises

- RuntimeError (Argument ‘depth‘ is not valid)
- RuntimeError (Argument ‘error‘ is not valid)
- RuntimeError (Argument ‘exclude‘ is not valid)
- RuntimeError (Argument ‘raised‘ is not valid)
- RuntimeError (Argument ‘width‘ is not valid)
- RuntimeError (Callable not found in exception list: *[name]*)
- RuntimeError (Unable to determine callable name)

get_sphinx_doc(*name*, *depth=None*, *exclude=None*, *width=72*, *error=False*, *raised=False*)
Returns an exception list marked up in reStructuredText

Parameters

- **name** (*string*) – Name of the callable (method, function or class property) to generate exceptions documentation for
- **depth** (*non-negative integer or None*) – Hierarchy levels to include in the exceptions list (overrides default **depth** argument; see `putil.exdoc.ExDoc.depth`). If None exceptions at all depths are included
- **exclude** (*list of strings or None*) – List of (potentially partial) module and callable names to exclude from exceptions list (overrides default **exclude** argument; see `putil.exdoc.ExDoc.exclude`). If None all callables are included
- **width** (*integer*) – Maximum width of the lines of text (minimum 40)
- **error** (*boolean*) – Flag that indicates whether an exception should be raised if the callable is not found in the callables exceptions database (True) or not (False)
- **raised** (*boolean*) – Flag that indicates whether only exceptions that were raised (and presumably caught) should be documented (True) or all registered exceptions should be documented (False)

Raises

- RuntimeError (Argument ‘depth‘ is not valid)

- RuntimeError (Argument ‘error‘ is not valid)
- RuntimeError (Argument ‘exclude‘ is not valid)
- RuntimeError (Argument ‘raised‘ is not valid)
- RuntimeError (Argument ‘width‘ is not valid)
- RuntimeError (Callable not found in exception list: *[name]*)

depth

Gets or sets the default hierarchy levels to include in the exceptions per callable. For example, a function `my_func()` calls two other functions, `get_data()` and `process_data()`, and in turn `get_data()` calls another function, `open_socket()`. In this scenario, the calls hierarchy is:

```
my_func             <- depth = 0
get_data           <- depth = 1
|open_socket       <- depth = 2
process_data       <- depth = 1
```

Setting `depth=0` means that only exceptions raised by `my_func()` are going to be included in the documentation; Setting `depth=1` means that only exceptions raised by `my_func()`, `get_data()` and `process_data()` are going to be included in the documentation; and finally setting `depth=2` (in this case it has the same effects as `depth=None`) means that only exceptions raised by `my_func()`, `get_data()`, `process_data()` and `open_socket()` are going to be included in the documentation.

Return type non-negative integer

Raises RuntimeError (Argument ‘depth‘ is not valid)

exclude

Gets or sets the default list of (potentially partial) module and callable names to exclude from exceptions per callable. For example, `['putil.ex']` excludes all exceptions from modules `putil.exh` and `putil.exdoc` (it acts as `r'putil.ex*'`). In addition to these modules, `['putil.ex', 'putil.eng.peng']` excludes exceptions from the function `putil.eng.peng`.

Return type list

Raises RuntimeError (Argument ‘exclude‘ is not valid)

6.9 exh module

This module can be used to register exceptions and then raise them if a given condition is true. For example:

```
# exh_example.py
from __future__ import print_function
import putil.exh

EXHOBJ = putil.exh.ExHandle()

def my_func(name):
    """ Sample function """
    EXHOBJ.add_exception(
        exname='illegal_name',
        extype=TypeError,
        exmsg='Argument `name` is not valid'
    )
    EXHOBJ.raise_exception_if(
```

```
        exname='illegal_name',
        condition=not isinstance(name, str)
    )
    print('My name is {0}'.format(name))
```

```
>>> import docs.support.exh_example
>>> docs.support.exh_example.my_func('Tom')
My name is Tom
>>> docs.support.exh_example.my_func(5)
Traceback (most recent call last):
    ...
TypeError: Argument `name` is not valid
```

When `my_func()` gets called with anything but a string as an argument a `TypeError` exception is raised with the message `'Argument `name` is not valid'`. While adding/registering an exception with *putil.exh.ExHandle.add_exception()* and conditionally raising it with *putil.exh.ExHandle.raise_exception_if()* takes the same number of lines of code as an exception raised inside an `if` block and incurs a slight performance penalty, using the *exh module* allows for automatic documentation of the exceptions raised by any function, method or class property with the help of the *exdoc module*.

6.9.1 Functions

putil.exh.**get_exh_obj**()
Returns the global exception handler

Return type *putil.exh.ExHandle* if global exception handler is set, None otherwise

putil.exh.**get_or_create_exh_obj**(*full_cname=False*, *exclude=None*, *callables_fname=None*)
Returns the global exception handler if it is set, otherwise creates a new global exception handler and returns it

Parameters

- **full_cname** (*boolean*) – Flag that indicates whether fully qualified function/method/class property names are obtained for functions/methods/class properties that use the exception manager (True) or not (False).

  There is a performance penalty if the flag is True as the call stack needs to be traced. This argument is only relevant if the global exception handler is not set and a new one is created
- **exclude** (*list of strings or None*) – Module exclusion list. A particular callable in an otherwise fully qualified name is omitted if it belongs to a module in this list. If None all callables are included
- **callables_fname** (*FileNameExists* or None) – File name that contains traced modules information. File can be produced by either the *putil.pinspect.Callables.save()* or *putil.exh.ExHandle.save_callables()* methods

Return type *putil.exh.ExHandle*

Raises

- OSError (File *[callables_fname]* could not be found
- RuntimeError (Argument ‘exclude‘ is not valid)
- RuntimeError (Argument ‘callables_fname‘ is not valid)
- RuntimeError (Argument ‘full_cname‘ is not valid)

putil.exh.**del_exh_obj**()
Deletes global exception handler (if set)

putil.exh.**set_exh_obj**(*obj*)
Sets the global exception handler

Parameters **obj** (*putil.exh.ExHandle*) – Exception handler

Raises RuntimeError (Argument 'obj' is not valid)

6.9.2 Classes

class putil.exh.**ExHandle**(*full_cname=False*, *exclude=None*, *callables_fname=None*)
Bases: object

Exception handler

Parameters

- **full_cname** (*boolean*) – Flag that indicates whether fully qualified function/method/class property names are obtained for functions/methods/class properties that use the exception manager (True) or not (False).

  There is a performance penalty if the flag is True as the call stack needs to be traced
- **exclude** (*list of strings or None*) – Module exclusion list. A particular callable in an otherwise fully qualified name is omitted if it belongs to a module in this list. If None all callables are included
- **callables_fname** (*FileNameExists* or None) – File name that contains traced modules information. File can be produced by either the *putil.pinspect.Callables.save()* or *putil.exh.ExHandle.save_callables()* methods

Return type *putil.exh.ExHandle*

Raises

- OSError (File *[callables_fname]* could not be found
- RuntimeError (Argument 'exclude' is not valid)
- RuntimeError (Argument 'callables_fname' is not valid)
- RuntimeError (Argument 'full_cname' is not valid)
- ValueError (Source for module *[module_name]* could not be found)

__add__(*other*)
Merges two objects.

Raises

- RuntimeError (Incompatible exception handlers)
- TypeError (Unsupported operand type(s) for +: putil.exh.ExHandle and *[other_type]*)

For example:

```
>>> import copy, putil.exh, putil.eng, putil.tree
>>> exhobj = putil.exh.set_exh_obj(putil.exh.ExHandle())
>>> putil.eng.peng(100, 3, True)
' 100.000 '
>>> tobj = putil.tree.Tree().add_nodes([{'name':'a', 'data':5}])
```

```
>>> obj1 = copy.copy(putil.exh.get_exh_obj())
>>> putil.exh.del_exh_obj()
>>> exhobj = putil.exh.get_or_create_exh_obj()
>>> putil.eng.peng(100, 3, True) # Trace some exceptions
' 100.000 '
>>> obj2 = copy.copy(putil.exh.get_exh_obj())
>>> putil.exh.del_exh_obj()
>>> exhobj = putil.exh.get_or_create_exh_obj()
>>> tobj = putil.tree.Tree().add_nodes([{'name':'a', 'data':5}])
>>> obj3 = copy.copy(putil.exh.get_exh_obj())
>>> obj1 == obj2
False
>>> obj1 == obj3
False
>>> obj1 == obj2+obj3
True
```

__bool__()

Returns `False` if exception handler does not have any exception defined, `True` otherwise.

Note: This method applies to Python 3.x

For example:

```
>>> from __future__ import print_function
>>> import putil.exh
>>> obj = putil.exh.ExHandle()
>>> if obj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: False
>>> def my_func(exhobj):
...     exhobj.add_exception('test', RuntimeError, 'Message')
>>> my_func(obj)
>>> if obj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: True
```

__copy__()

Copies object. For example:

```
>>> import copy, putil.exh, putil.eng
>>> exhobj = putil.exh.set_exh_obj(putil.exh.ExHandle())
>>> putil.eng.peng(100, 3, True)
' 100.000 '
>>> obj1 = putil.exh.get_exh_obj()
>>> obj2 = copy.copy(obj1)
>>> obj1 == obj2
True
```

__eq__(*other*)

Tests object equality. For example:

```
>>> import copy, putil.exh, putil.eng
>>> exhobj = putil.exh.set_exh_obj(putil.exh.ExHandle())
>>> putil.eng.peng(100, 3, True)
```

```
' 100.000 '
>>> obj1 = putil.exh.get_exh_obj()
>>> obj2 = copy.copy(obj1)
>>> obj1 == obj2
True
>>> 5 == obj1
False
```

__iadd__(*other*)

Merges an object into an existing object.

Raises

- RuntimeError (Incompatible exception handlers)
- TypeError (Unsupported operand type(s) for +: putil.exh.ExHandle and *[other_type]*)

For example:

```
>>> import copy, putil.exh, putil.eng, putil.tree
>>> exhobj = putil.exh.set_exh_obj(putil.exh.ExHandle())
>>> putil.eng.peng(100, 3, True)
' 100.000 '
>>> tobj = putil.tree.Tree().add_nodes([{'name':'a', 'data':5}])
>>> obj1 = copy.copy(putil.exh.get_exh_obj())
>>> putil.exh.del_exh_obj()
>>> exhobj = putil.exh.get_or_create_exh_obj()
>>> putil.eng.peng(100, 3, True) # Trace some exceptions
' 100.000 '
>>> obj2 = copy.copy(putil.exh.get_exh_obj())
>>> putil.exh.del_exh_obj()
>>> exhobj = putil.exh.get_or_create_exh_obj()
>>> tobj = putil.tree.Tree().add_nodes([{'name':'a', 'data':5}])
>>> obj3 = copy.copy(putil.exh.get_exh_obj())
>>> obj1 == obj2
False
>>> obj1 == obj3
False
>>> obj2 += obj3
>>> obj1 == obj2
True
```

__nonzero__()

Returns `False` if exception handler does not have any exception defined, `True` otherwise.

Note: This method applies to Python 2.7

For example:

```
>>> from __future__ import print_function
>>> import putil.exh
>>> obj = putil.exh.ExHandle()
>>> if obj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: False
>>> def my_func(exhobj):
...     exhobj.add_exception('test', RuntimeError, 'Message')
>>> my_func(obj)
```

```
>>> if obj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: True
```

__str__()
Returns a string with a detailed description of the object's contents. For example:

```
>>> from __future__ import print_function
>>> import docs.support.exh_example
>>> docs.support.exh_example.my_func('Tom')
My name is Tom
>>> print(str(docs.support.exh_example.EXHOBJ))
Name    : .../illegal_name
Type    : TypeError
Message : Argument `name` is not valid
Function: None
```

add_exception(*exname*, *extype*, *exmsg*)
Adds an exception to the handler

Parameters

- **exname** (*string*) – Exception name; has to be unique within the namespace, duplicates are eliminated
- **extype** (*Exception type object, i.e. RuntimeError, TypeError, etc.*) – Exception type; *must* be derived from the Exception class
- **exmsg** (*string*) – Exception message; it can contain fields to be replaced when the exception is raised via *putil.exh.ExHandle.raise_exception_if()*. A field starts with the characters '*[' and ends with the characters ']*', the field name follows the same rules as variable names and is between these two sets of characters. For example, *[fname]* defines the fname field

Raises

- RuntimeError (Argument 'exmsg' is not valid)
- RuntimeError (Argument 'exname' is not valid)
- RuntimeError (Argument 'extype' is not valid)

raise_exception_if(*exname*, *condition*, *edata=None*)
Raises exception conditionally

Parameters

- **exname** (*string*) – Exception name
- **condition** (*boolean*) – Flag that indicates whether the exception is raised *(True)* or not *(False)*
- **edata** (*dictionary, iterable of dictionaries or None*) – Replacement values for fields in the exception message (see *putil.exh.ExHandle.add_exception()* for how to define fields). Each dictionary entry can only have these two keys:
  - **field** *(string)* – Field name
  - **value** *(any)* – Field value, to be converted into a string with the format string method

  If None no field replacement is done

Raises

- RuntimeError (Argument ‘condition‘ is not valid)
- RuntimeError (Argument ‘edata‘ is not valid)
- RuntimeError (Argument ‘exname‘ is not valid)
- RuntimeError (Field *[field_name]* not in exception message)
- ValueError (Exception name *[name]* not found’)

save_callables(*callables_fname*)
Saves traced modules information to a JSON file. If the file exists it is overwritten

Parameters **callables_fname** (*FileName*) – File name

Raises RuntimeError (Argument ‘callables_fname‘ is not valid)

callables_db
Returns the callables database of the modules using the exception handler, as reported by *putil.pinspect.Callables.callables_db()*

callables_separator
Returns the character (`'/'`) used to separate the sub-parts of fully qualified function names in *putil.exh.ExHandle.callables_db()* and **name** key in *putil.exh.ExHandle.exceptions_db()*

exceptions_db
Returns the exceptions database. This database is a list of dictionaries that contain the following keys:

•**name** *(string)* – Exception name of the form `'[callable_identifier]/[exception_name]'`. The contents of `[callable_identifier]` depend on the value of the argument **full_cname** used to create the exception handler.

If **full_cname** is True, `[callable_identifier]` is the fully qualified callable name as it appears in the callables database (*putil.exh.ExHandle.callables_db()*).

If **full_cname** is False, then `[callable_identifier]` is a decimal string representation of the callable's code identifier as reported by the id() function.

In either case `[exception_name]` is the name of the exception provided when it was defined in *putil.exh.ExHandle.add_exception()* (**exname** argument)

•**data** *(string)* – Text of the form `'[exception_type]([exception_message])[raised]'` where `[exception_type]` and `[exception_message]` are the exception type and exception message, respectively, given when the exception was defined by *putil.exh.ExHandle.add_exception()* (**extype** and **exmsg** arguments); and `raised` is an asterisk (`'*'`) when the exception has been raised via *putil.exh.ExHandle.raise_exception_if()*, the empty string (`''`) otherwise

6.10 misc module

This module contains miscellaneous utility functions that can be applied in a variety of circumstances; there are context managers, membership functions (test if an argument is of a given type), numerical functions and string functions

6.10.1 Context managers

putil.misc.**ignored**(**exceptions*)

Executes commands and selectively ignores exceptions (Inspired by "Transforming Code into Beautiful, Idiomatic Python" talk at PyCon US 2013 by Raymond Hettinger)

Parameters **exceptions** (*Exception object, i.e. RuntimeError, OSError, etc.*) – Exception type(s) to ignore

For example:

```
# misc_example_1.py
from __future__ import print_function
import os, putil.misc

def ignored_example():
    fname = 'somefile.tmp'
    open(fname, 'w').close()
    print('File {0} exists? {1}'.format(
        fname, os.path.isfile(fname)
    ))
    with putil.misc.ignored(OSError):
        os.remove(fname)
    print('File {0} exists? {1}'.format(
        fname, os.path.isfile(fname)
    ))
    with putil.misc.ignored(OSError):
        os.remove(fname)
    print('No exception trying to remove a file that does not exists')
    try:
        with putil.misc.ignored(RuntimeError):
            os.remove(fname)
    except:
        print('Got an exception')
```

```
>>> import docs.support.misc_example_1
>>> docs.support.misc_example_1.ignored_example()
File somefile.tmp exists? True
File somefile.tmp exists? False
No exception trying to remove a file that does not exists
Got an exception
```

class putil.misc.**Timer**(*verbose=False*)

Bases: object

Profiles blocks of code by calculating elapsed time between the context manager entry and exit time points. Inspired by Huy Nguyen's blog

Parameters **verbose** (*boolean*) – Flag that indicates whether the elapsed time is printed upon exit (True) or not (False)

Returns *putil.misc.Timer*

Raises RuntimeError (Argument 'verbose' is not valid)

For example:

```
# misc_example_2.py
from __future__ import print_function
import putil.misc
```

```
def timer(num_tries, fpointer):
    with putil.misc.Timer() as tobj:
        for _ in range(num_tries):
            fpointer()
    print('Time per call: {0} seconds'.format(
        tobj.elapsed_time/(2.0*num_tries)
    ))

def sample_func():
    count = 0
    for num in range(0, count):
        count += num
```

```
>>> from docs.support.misc_example_2 import *
>>> timer(100, sample_func)
Time per call: ... seconds
```

elapsed_time
Returns elapsed time (in seconds) between context manager entry and exit time points

Return type float

class putil.misc.**TmpFile**(*fpointer=None*)
Bases: object

Creates a temporary file and optionally sets up hooks for a function to write data to it

Parameters **fpointer** (*function object or None*) – Pointer to a function that writes data to file. If the argument is not None the function pointed to receives exactly one argument, a file-like object as created by the tempfile.NamedTemporaryFile function

Returns temporary file name

Raises RuntimeError (Argument ‘fpointer‘ is not valid)

For example:

```
# misc_example_3.py
from __future__ import print_function
import sys, putil.misc

def write_data(file_handle):
    if sys.hexversion < 0x03000000:
        file_handle.write('Hello world!')
    else:
        file_handle.write(bytes('Hello world!', 'ascii'))

def show_tmpfile():
    with putil.misc.TmpFile(write_data) as fname:
        with open(fname, 'r') as fobj:
            lines = fobj.readlines()
    print('\n'.join(lines))
```

```
>>> from docs.support.misc_example_3 import *
>>> show_tmpfile()
Hello world!
```

6.10.2 File

putil.misc.**make_dir**(*fname*)
Creates the directory of a fully qualified file name if it does not exist

Parameters **fname** (*string*) – File name

Equivalent to these Bash shell commands:

```
$ dir=$(dirname ${fname})
$ mkdir -p ${dir}
```

Parameters **fname** (*string*) – Fully qualified file name

6.10.3 Membership

putil.misc.**isalpha**(*obj*)
Tests if the argument is a string representing a number

Parameters **obj** (*any*) – Object

Return type boolean

For example:

```
>>> import putil.misc
>>> putil.misc.isalpha('1.5')
True
>>> putil.misc.isalpha('1E-20')
True
>>> putil.misc.isalpha('1EA-20')
False
```

putil.misc.**ishex**(*obj*)
Tests if the argument is a string representing a valid hexadecimal digit

Parameters **obj** (*any*) – Object

Return type boolean

putil.misc.**isiterable**(*obj*)
Tests if the argument is an iterable

Parameters **obj** (*any*) – Object

Return type boolean

putil.misc.**isnumber**(*obj*)
Tests if the argument is a number (complex, float or integer)

Parameters **obj** (*any*) – Object

Return type boolean

putil.misc.**isreal**(*obj*)
Tests if the argument is a real number (float or integer)

Parameters **obj** (*any*) – Object

Return type boolean

6.10.4 Miscellaneous

putil.misc.**flatten_list**(*lobj*)
Recursively flattens a list

Parameters **lobj** (*list*) – List to flatten

Return type list

For example:

```
>>> import putil.misc
>>> putil.misc.flatten_list([1, [2, 3, [4, 5, 6]], 7])
[1, 2, 3, 4, 5, 6, 7]
```

putil.misc.**pprint_ast_node**(*node*, *annotate_fields=True*, *include_attributes=False*, *indent=' '*)
Emulates the AST module dump function but with prettier printing. From Alex Leone's blog

Parameters

- **node** (*AST object*) – root abstract syntax tree node
- **annotate_fields** (*boolean*) – Flag that indicates whether name and values for fields are shown (True) or not (False); the latter is required if code is to be evaluated
- **include_attributes** (*boolean*) – Flag that indicates whether line numbers and column offsets are dumped (True) or not (False)
- **indent** (*string*) – Characters to use for indenting output sub-nodes and structures

Return type string

Raises RuntimeError (Argument 'node' is not valid)

6.10.5 Numbers

putil.misc.**gcd**(*vector*)
Calculates the greatest common divisor (GCD) of a list of numbers or a Numpy vector of numbers. The computations are carried out with a precision of 1E-12 if the objects are not fractions. When possible it is best to use the fractions data type with the numerator and denominator arguments when computing the GCD of floating point numbers.

Parameters **vector** (*list of numbers or Numpy vector of numbers*) – Vector of numbers

putil.misc.**normalize**(*value*, *series*, *offset=0*)
Scales a value to the range defined by a series

Parameters

- **value** (*number*) – Value to normalize
- **series** (*list*) – List of numbers that defines the normalization range
- **offset** (*number*) – Normalization offset, i.e. the returned value will be in the range [**offset**, 1.0]

Return type number

Raises

- RuntimeError (Argument 'offset' is not valid)
- RuntimeError (Argument 'series' is not valid)

- RuntimeError (Argument ‘value‘ is not valid)
- ValueError (Argument ‘offset‘ has to be in the [0.0, 1.0] range)
- ValueError (Argument ‘value‘ has to be within the bounds of the argument ‘series‘)

For example:

```
>>> import putil.misc
>>> putil.misc.normalize(15, [10, 20])
0.5
>>> putil.misc.normalize(15, [10, 20], 0.5)
0.75
```

putil.misc.**per**(*arga*, *argb*, *prec=10*)

Calculates the percentage difference between two numbers or the element-wise percentage difference between two lists of numbers or Numpy vectors. If any of the numbers in the arguments is zero the value returned is 1E+20

Parameters

- **arga** (*float, integer, list of floats or integers, or Numpy vector of floats or integers*) – First number, list of numbers or Numpy vector
- **argb** (*float, integer, list of floats or integers, or Numpy vector of floats or integers*) – Second number, list of numbers or or Numpy vector
- **prec** (*integer*) – Maximum length of the fractional part of the result

Return type Float, list of floats or Numpy vector, depending on the arguments type

Raises

- RuntimeError (Argument ‘arga‘ is not valid)
- RuntimeError (Argument ‘argb‘ is not valid)
- RuntimeError (Argument ‘prec‘ is not valid)
- TypeError (Arguments are not of the same type)

putil.misc.**pgcd**(*numa*, *numb*)

Calculate the greatest common divisor (GCD) of two numbers

Parameters

- **numa** (*number*) – First number
- **numb** (*number*) – Second number

Return type number

For example:

```
>>> import putil.misc, fractions
>>> putil.misc.pgcd(10, 15)
5
>>> str(putil.misc.pgcd(0.05, 0.02))
'0.01'
>>> str(putil.misc.pgcd(5/3.0, 2/3.0))[:6]
'0.3333'
>>> putil.misc.pgcd(
...     fractions.Fraction(str(5/3.0)),
...     fractions.Fraction(str(2/3.0))
... )
Fraction(1, 3)
```

```
>>> putil.misc.pgcd(
...     fractions.Fraction(5, 3),
...     fractions.Fraction(2, 3)
... )
Fraction(1, 3)
```

6.10.6 String

putil.misc.**binary_string_to_octal_string**(*text*)

Returns a binary-packed string in octal representation aliasing typical codes to their escape sequences

Parameters **text** (*string*) – Text to convert

Return type string

Code	Alias	Description
0	\0	Null character
7	\a	Bell / alarm
8	\b	Backspace
9	\t	Horizontal tab
10	\n	Line feed
11	\v	Vertical tab
12	\f	Form feed
13	\r	Carriage return

For example:

```
>>> import putil.misc, struct, sys
>>> def py23struct(num):
...     if sys.hexversion < 0x03000000:
...         return struct.pack('h', num)
...     else:
...         return struct.pack('h', num).decode('ascii')
>>> nums = range(1, 15)
>>> putil.misc.binary_string_to_octal_string(
...     ''.join([py23struct(num) for num in nums])
... ).replace('o', '')
'\\1\\0\\2\\0\\3\\0\\4\\0\\5\\0\\6\\0\\a\\0\\b\\0\\t\\0\\...
```

putil.misc.**char_to_decimal**(*text*)

Converts a string to its decimal ASCII representation, with spaces between characters

Parameters **text** (*string*) – Text to convert

Return type string

For example:

```
>>> import putil.misc
>>> putil.misc.char_to_decimal('Hello world!')
'72 101 108 108 111 32 119 111 114 108 100 33'
```

putil.misc.**elapsed_time_string**(*start_time*, *stop_time*)

Returns a formatted string with the elapsed time between two time points. The string includes years (365 days), months (30 days), days (24 hours), hours (60 minutes), minutes (60 seconds) and seconds. If both arguments are equal, the string returned is `'None'`; otherwise, the string returned is [YY year[s], [MM month[s], [DD day[s], [HH hour[s], [MM minute[s] [and SS second[s]]]]]]. Any part (year[s], month[s], etc.) is omitted if the value of that part is null/zero

Parameters

- **start_time** (datetime) – Starting time point
- **stop_time** (*datetime*) – Ending time point

Return type string

Raises RuntimeError (Invalid time delta specification)

For example:

```
>>> import datetime, putil.misc
>>> start_time = datetime.datetime(2014, 1, 1, 1, 10, 1)
>>> stop_time = datetime.datetime(2015, 1, 3, 1, 10, 3)
>>> putil.misc.elapsed_time_string(start_time, stop_time)
'1 year, 2 days and 2 seconds'
```

putil.misc.**pcolor**(*text*, *color*, *indent=0*)

Returns a string that once printed is colorized

Parameters

- **text** (*string*) – Text to colorize
- **color** (*string*) – Color to use, one of `'black'`, `'red'`, `'green'`, `'yellow'`, `'blue'`, `'magenta'`, `'cyan'`, `'white'` or `'none'` (case insensitive)
- **indent** (*integer*) – Number of spaces to prefix the output with

Return type string

Raises

- RuntimeError (Argument ‘color‘ is not valid)
- RuntimeError (Argument ‘indent‘ is not valid)
- RuntimeError (Argument ‘text‘ is not valid)
- ValueError (Unknown color *[color]*)

putil.misc.**quote_str**(*obj*)

Adds extra quotes to a string. If the argument is not a string it is returned unmodified

Parameters **obj** (*any*) – Object

Return type Same as argument

For example:

```
>>> import putil.misc
>>> putil.misc.quote_str(5)
5
>>> putil.misc.quote_str('Hello!')
'"Hello!"'
>>> putil.misc.quote_str('He said "hello!"')
'\'He said "hello!"\''
```

putil.misc.**strframe**(*obj*, *extended=False*)

Returns a string with a frame record (typically an item in a list generated by inspect.stack()) pretty printed

Parameters

- **obj** (*tuple*) – Frame record

- **extended** (*boolean*) – Flag that indicates whether contents of the frame object are printed (True) or not (False)

Return type string

6.11 pcsv module

This module can be used to handle comma-separated values (CSV) files and do lightweight processing of their data with support for row and column filtering. In addition to basic read, write and data replacement, files can be concatenated, merged, and sorted

6.11.1 Examples

Read/write

```
# pcsv_example_1.py
import putil.pcsv, tempfile

def main():
    with tempfile.NamedTemporaryFile() as fobj:
        fname = fobj.name
        ref_data = [
            ['Item', 'Cost'],
            [1, 9.99],
            [2, 10000],
            [3, 0.10]
        ]
        # Write reference data to a file
        putil.pcsv.write(fname, ref_data, append=False)
        # Read the data back
        obj = putil.pcsv.CsvFile(fname)
    # After the object creation the I/O is done,
    # can safely remove file (exit context manager)
    # Check that data read is correct
    assert obj.header() == ref_data[0]
    assert obj.data() == ref_data[1:]
    # Add a simple row filter, only look at rows that have
    # values 1 and 3 in the "Items" column
    obj.rfilter = {'Item':[1, 3]}
    assert obj.data(filtered=True) == [ref_data[1], ref_data[3]]

if __name__ == '__main__':
    main()
```

Replace data

```
# pcsv_example_2.py
import putil.pcsv, tempfile

def main():
    ctx = tempfile.NamedTemporaryFile
    with ctx() as ifobj:
        with ctx() as rfobj:
```

```
            with ctx() as ofobj:
                # Create first (input) data file
                input_data = [
                    ['Item', 'Cost'],
                    [1, 9.99],
                    [2, 10000],
                    [3, 0.10]
                ]
                ifname = ifobj.name
                putil.pcsv.write(ifname, input_data, append=False)
                # Create second (replacement) data file
                replacement_data = [
                    ['Staff', 'Rate', 'Days'],
                    ['Joe', 10, 'Sunday'],
                    ['Sue', 20, 'Thursday'],
                    ['Pat', 15, 'Tuesday']
                ]
                rfname = rfobj.name
                putil.pcsv.write(rfname, replacement_data, append=False)
                # Replace "Cost" column of input file with "Rate" column
                # of replacement file for "Items" 2 and 3 with "Staff" data
                # from Joe and Pat. Save resulting data to another file
                ofname = ofobj.name
                putil.pcsv.replace(
                    ifname=ifname,
                    idfilter=('Cost', {'Item':[1, 3]}),
                    rfname=rfname,
                    rdfilter=('Rate', {'Staff':['Joe', 'Pat']}),
                    ofname=ofname
                )
                # Verify that resulting file is correct
                ref_data = [
                    ['Item', 'Cost'],
                    [1, 10],
                    [2, 10000],
                    [3, 15]
                ]
                obj = putil.pcsv.CsvFile(ofname)
                assert obj.header() == ref_data[0]
                assert obj.data() == ref_data[1:]

if __name__ == '__main__':
    main()
```

Concatenate two files

```
# pcsv_example_3.py
import putil.pcsv, tempfile

def main():
    ctx = tempfile.NamedTemporaryFile
    with ctx() as fobj1:
        with ctx() as fobj2:
            with ctx() as ofobj:
                # Create first data file
                data1 = [
                    [1, 9.99],
```

```
                    [2, 10000],
                    [3, 0.10]
                ]
                fname1 = fobj1.name
                putil.pcsv.write(fname1, data1, append=False)
                # Create second data file
                data2 = [
                    ['Joe', 10, 'Sunday'],
                    ['Sue', 20, 'Thursday'],
                    ['Pat', 15, 'Tuesday']
                ]
                fname2 = fobj2.name
                putil.pcsv.write(fname2, data2, append=False)
                # Concatenate file1 and file2. Filter out
                # second column of file2
                ofname = ofobj.name
                putil.pcsv.concatenate(
                    fname1=fname1,
                    fname2=fname2,
                    has_header1=False,
                    has_header2=False,
                    dfilter2=[0, 2],
                    ofname=ofname,
                    ocols=['D1', 'D2']
                )
                # Verify that resulting file is correct
                ref_data = [
                    ['D1', 'D2'],
                    [1, 9.99],
                    [2, 10000],
                    [3, 0.10],
                    ['Joe', 'Sunday'],
                    ['Sue', 'Thursday'],
                    ['Pat', 'Tuesday']
                ]
                obj = putil.pcsv.CsvFile(ofname)
                assert obj.header() == ref_data[0]
                assert obj.data() == ref_data[1:]

if __name__ == '__main__':
    main()
```

Merge two files

```
# pcsv_example_4.py
import putil.pcsv, tempfile

def main():
    ctx = tempfile.NamedTemporaryFile
    with ctx() as fobj1:
        with ctx() as fobj2:
            with ctx() as ofobj:
                # Create first data file
                data1 = [
                    [1, 9.99],
                    [2, 10000],
                    [3, 0.10]
```

```
                ]
                fname1 = fobj1.name
                putil.pcsv.write(fname1, data1, append=False)
                # Create second data file
                data2 = [
                    ['Joe', 10, 'Sunday'],
                    ['Sue', 20, 'Thursday'],
                    ['Pat', 15, 'Tuesday']
                ]
                fname2 = fobj2.name
                putil.pcsv.write(fname2, data2, append=False)
                # Merge file1 and file2
                ofname = ofobj.name
                putil.pcsv.merge(
                    fname1=fname1,
                    has_header1=False,
                    fname2=fname2,
                    has_header2=False,
                    ofname=ofname
                )
                # Verify that resulting file is correct
                ref_data = [
                    [1, 9.99, 'Joe', 10, 'Sunday'],
                    [2, 10000, 'Sue', 20, 'Thursday'],
                    [3, 0.10, 'Pat', 15, 'Tuesday'],
                ]
                obj = putil.pcsv.CsvFile(ofname, has_header=False)
                assert obj.header() == list(range(0, 5))
                assert obj.data() == ref_data

if __name__ == '__main__':
    main()
```

Sort a file

```
# pcsv_example_5.py
import putil.pcsv, tempfile

def main():
    ctx = tempfile.NamedTemporaryFile
    with ctx() as ifobj:
        with ctx() as ofobj:
            # Create first data file
            data = [
                ['Ctrl', 'Ref', 'Result'],
                [1, 3, 10],
                [1, 4, 20],
                [2, 4, 30],
                [2, 5, 40],
                [3, 5, 50]
            ]
            ifname = ifobj.name
            ofname = ofobj.name
            putil.pcsv.write(ifname, data, append=False)
            # Sort
            putil.pcsv.dsort(
                fname=ifname,
```

```
                order=[{'Ctrl':'D'}, {'Ref':'A'}],
                has_header=True,
                ofname=ofname
            )
            # Verify that resulting file is correct
            ref_data = [
                [3, 5, 50],
                [2, 4, 30],
                [2, 5, 40],
                [1, 3, 10],
                [1, 4, 20]
            ]
            obj = putil.pcsv.CsvFile(ofname, has_header=True)
            assert obj.header() == ['Ctrl', 'Ref', 'Result']
            assert obj.data() == ref_data

if __name__ == '__main__':
    main()
```

6.11.2 Identifying (filtering) columns

Several class methods and functions in this module allow column and row filtering of the CSV file data. It is necessary to identify columns for both of these operations and how these columns can be identified depends on whether the file has or does not have a header as indicated by the **has_header** boolean constructor argument:

- If **has_header** is `True` the first line of the file is taken as the header. Columns can be identified by name (a string that has to match a column value in the file header) or by number (an integer representing the column number with column zero being the leftmost column)
- If **has_header** is `False` columns can only be identified by number (an integer representing the column number with column zero being the leftmost column)

For example, if a file `myfile.csv` has the following data:

Ctrl	Ref	Result
1	3	10
1	4	20
2	4	30
2	5	40
3	5	50

Then when the file is loaded with `putil.pcsv.CsvFile('myfile.csv', has_header=True)` the columns can be referred to as `'Ctrl'` or `0`, `'Ref'` or `1`, or `'Result'` or `2`. However if the file is loaded with `putil.pcsv.CsvFile('myfile.csv', has_header=False)` the columns can be referred only as `0`, `1` or `2`.

6.11.3 Filtering rows

Several class methods and functions of this module allow row filtering of the CSV file data. The row filter is described in the *CsvRowFilter* pseudo-type

6.11.4 Swapping or inserting columns

The column filter not only filters columns *but also* determines the order in which the columns are stored internally in an *putil.pcsv.CsvFile* object. This means that the column filter can be used to reorder and/or duplicate

columns. For example:

```
# pcsv_example_6.py
import putil.pcsv, tempfile

def main():
    ctx = tempfile.NamedTemporaryFile
    with ctx() as ifobj:
        with ctx() as ofobj:
            ifname = ifobj.name
            ofname = ofobj.name
            # Create input data file
            data = [
                ['Ctrl', 'Ref', 'Result'],
                [1, 3, 10],
                [1, 4, 20],
                [2, 4, 30],
                [2, 5, 40],
                [3, 5, 50]
            ]
            putil.pcsv.write(ifname, data, append=False)
            # Swap 'Ctrl' and 'Result' columns, duplicate
            # 'Ref' column at the end
            obj = putil.pcsv.CsvFile(
                fname=ifname,
                dfilter=['Result', 'Ref', 'Ctrl', 1],
            )
            assert obj.header(filtered=False) == ['Ctrl', 'Ref', 'Result']
            assert (
                obj.header(filtered=True)
                ==
                ['Result', 'Ref', 'Ctrl', 'Ref']
            )
            obj.write(
                ofname,
                header=['Result', 'Ref', 'Ctrl', 'Ref2'],
                filtered=True,
                append=False
            )
            # Verify that resulting file is correct
            ref_data = [
                [10, 3, 1, 3],
                [20, 4, 1, 4],
                [30, 4, 2, 4],
                [40, 5, 2, 5],
                [50, 5, 3, 5]
            ]
            obj = putil.pcsv.CsvFile(ofname, has_header=True)
            assert obj.header() == ['Result', 'Ref', 'Ctrl', 'Ref2']
            assert obj.data() == ref_data

if __name__ == '__main__':
    main()
```

6.11.5 Empty columns

When a file has empty columns they are read as `None`. Conversely any column value that is `None` is written as an empty column. Empty columns are ones that have either an empty string (`''`) or literally no information between the column delimiters (`,`)

For example, if a file `myfile2.csv` has the following data:

Ctrl	Ref	Result
1	4	20
2		30
2	5	
	5	50

The corresponding read array is:

```
[
    ['Ctrl', 'Ref', 'Result'],
    [1, 4, 20],
    [2, None, 30],
    [2, 5, None],
    [None, 5, 50]
]
```

6.11.6 Functions

putil.pcsv.**concatenate**(*fname1*, *fname2*, *dfilter1=None*, *dfilter2=None*, *has_header1=True*, *has_header2=True*, *ofname=None*, *ocols=None*)

Concatenates two comma-separated values file. Data rows from the second file are appended at the end of the data rows from the first file

Parameters

- **fname1** (*FileNameExists*) – Name of the first comma-separated values file, the file whose data appears first in the output file
- **fname2** (*FileNameExists*) – Name of the second comma-separated values file, the file whose data appears last in the output file
- **dfilter1** (*CsvDataFilter* or None) – Row and/or column filter for the first file. If None no data filtering is done on the file
- **dfilter2** (*CsvDataFilter* or None) – Row and/or column filter for the second file. If None no data filtering is done on the file
- **has_header1** (*boolean*) – Flag that indicates whether the first comma-separated values file has column headers in its first line (True) or not (False)
- **has_header2** (*boolean*) – Flag that indicates whether the second comma-separated values file has column headers in its first line (True) or not (False)
- **ofname** (*FileName* or None) – Name of the output comma-separated values file, the file that will contain the data from the first and second files. If None the first file is replaced "in place"
- **ocols** (*list or None*) – Column names of the output comma-separated values file. If None the column names in the first file are used if **has_header1** is True or the column names in the second files are used if **has_header1** is False and **has_header2** is True, otherwise no header is used

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument ‘dfilter1‘ is not valid)
- RuntimeError (Argument ‘dfilter2‘ is not valid)
- RuntimeError (Argument ‘fname1‘ is not valid)
- RuntimeError (Argument ‘fname2‘ is not valid)
- RuntimeError (Argument ‘ocols‘ is not valid)
- RuntimeError (Argument ‘ofname‘ is not valid)
- RuntimeError (Column headers are not unique in file *[fname]*)
- RuntimeError (File *[fname]* is empty)
- RuntimeError (Files have different number of columns)
- RuntimeError (Invalid column specification)
- RuntimeError (Number of columns in data files and output columns are different)
- ValueError (Column *[column_identifier]* not found)

putil.pcsv.**dsort**(*fname*, *order*, *has_header=True*, *ofname=None*)

Sorts file data

Parameters

- **fname** (*FileNameExists*) – Name of the comma-separated values file to sort
- **order** (*CsvColFilter*) – Sort order
- **has_header** (*boolean*) – Flag that indicates whether the comma-separated values file to sort has column headers in its first line (True) or not (False)
- **ofname** (*FileName* or None) – Name of the output comma-separated values file, the file that will contain the sorted data. If None the sorting is done “in place”

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument ‘fname‘ is not valid)
- RuntimeError (Argument ‘has_header‘ is not valid)
- RuntimeError (Argument ‘ofname‘ is not valid)
- RuntimeError (Argument ‘order‘ is not valid)
- RuntimeError (Column headers are not unique in file *[fname]*)
- RuntimeError (File *[fname]* is empty)
- RuntimeError (Invalid column specification)
- ValueError (Column *[column_identifier]* not found)

putil.pcsv.**merge**(*fname1*, *fname2*, *dfilter1=None*, *dfilter2=None*, *has_header1=True*, *has_header2=True*, *ofname=None*, *ocols=None*)

Merges two comma-separated values files. Data columns from the second file are appended after data columns from the first file. Empty values in columns are used if the files have different number of rows

Parameters

- **fname1** (*FileNameExists*) – Name of the first comma-separated values file, the file whose columns appear first in the output file
- **fname2** (*FileNameExists*) – Name of the second comma-separated values file, the file whose columns appear last in the output file
- **dfilter1** (*CsvDataFilter* or None) – Row and/or column filter for the first file. If None no data filtering is done on the file
- **dfilter2** (*CsvDataFilter* or None) – Row and/or column filter for the second file. If None no data filtering is done on the file
- **has_header1** (*boolean*) – Flag that indicates whether the first comma-separated values file has column headers in its first line (True) or not (False)
- **has_header2** (*boolean*) – Flag that indicates whether the second comma-separated values file has column headers in its first line (True) or not (False)
- **ofname** (*FileName* or None) – Name of the output comma-separated values file, the file that will contain the data from the first and second files. If None the first file is replaced "in place"
- **ocols** (*list or None*) – Column names of the output comma-separated values file. If None the column names in the first and second files are used if **has_header1** and/or **has_header2** are True. The column labels `'Column [column_number]'` are used when one of the two files does not have a header, where `[column_number]` is an integer representing the column number (column 0 is the leftmost column). No header is used if **has_header1** and **has_header2** are False

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument 'dfilter1' is not valid)
- RuntimeError (Argument 'dfilter2' is not valid)
- RuntimeError (Argument 'fname1' is not valid)
- RuntimeError (Argument 'fname2' is not valid)
- RuntimeError (Argument 'ocols' is not valid)
- RuntimeError (Argument 'ofname' is not valid)
- RuntimeError (Column headers are not unique in file *[fname]*)
- RuntimeError (Combined columns in data files and output columns are different)
- RuntimeError (File *[fname]* is empty)
- RuntimeError (Invalid column specification)
- ValueError (Column *[column_identifier]* not found)

putil.pcsv.**replace**(*ifname*, *idfilter*, *rfname*, *rdfilter*, *ihas_header=True*, *rhas_header=True*, *ofname=None*, *ocols=None*)

Replaces data in one file with data from another file

Parameters

- **ifname** (*FileNameExists*) – Name of the input comma-separated values file, the file that contains the columns to be replaced

- **idfilter** (*CsvDataFilter*) – Row and/or column filter for the input file
- **rfname** (*FileNameExists*) – Name of the replacement comma-separated values file, the file that contains the replacement data
- **rdfilter** (*CsvDataFilter*) – Row and/or column filter for the replacement file
- **ihas_header** (*boolean*) – Flag that indicates whether the input comma-separated values file has column headers in its first line (True) or not (False)
- **rhas_header** (*boolean*) – Flag that indicates whether the replacement comma-separated values file has column headers in its first line (True) or not (False)
- **ofname** (*FileName* or None) – Name of the output comma-separated values file, the file that will contain the input file data but with some columns replaced with data from the replacement file. If None the input file is replaced "in place"
- **ocols** (*list or None*) – Names of the replaced columns in the output comma-separated values file. If None the column names in the input file are used if **ihas_header** is True, otherwise no header is used

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument 'idfilter' is not valid)
- RuntimeError (Argument 'ifname' is not valid)
- RuntimeError (Argument 'ocols' is not valid)
- RuntimeError (Argument 'ofname' is not valid)
- RuntimeError (Argument 'rdfilter' is not valid)
- RuntimeError (Argument 'rfname' is not valid)
- RuntimeError (Column headers are not unique in file *[fname]*)
- RuntimeError (File *[fname]* is empty)
- RuntimeError (Invalid column specification)
- RuntimeError (Number of input and output columns are different)
- RuntimeError (Number of input and replacement columns are different)
- ValueError (Column *[column_identifier]* not found)
- ValueError (Number of rows mismatch between input and replacement data)

putil.pcsv.**write**(*fname*, *data*, *append=True*)

Writes data to a specified comma-separated values (CSV) file

Parameters

- **fname** (*FileName*) – Name of the comma-separated values file to be written
- **data** (*list*) – Data to write to the file. Each item in this argument should contain a sub-list corresponding to a row of data; each item in the sub-lists should contain data corresponding to a particular column
- **append** (*boolean*) – Flag that indicates whether data is added to an existing file (or a new file is created if it does not exist) (True), or whether data overwrites the file contents (if the file exists) or creates a new file if the file does not exists (False)

Raises

- OSError (File *[fname]* could not be created: *[reason]*)
- RuntimeError (Argument 'append' is not valid)
- RuntimeError (Argument 'data' is not valid)
- RuntimeError (Argument 'fname' is not valid)
- ValueError (There is no data to save to file)

6.11.7 Classes

class putil.pcsv.**CsvFile**(*fname*, *dfilter=None*, *has_header=True*)

Bases: object

Processes comma-separated values (CSV) files

Parameters

- **fname** (*FileNameExists*) – Name of the comma-separated values file to read
- **dfilter** (*CsvDataFilter* or None) – Row and/or column filter. If None no data filtering is done
- **has_header** (*boolean*) – Flag that indicates whether the comma-separated values file has column headers in its first line (True) o not (False)

Return type *putil.pcsv.CsvFile* object

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument 'dfilter' is not valid)
- RuntimeError (Argument 'fname' is not valid)
- RuntimeError (Argument 'has_header' is not valid)
- RuntimeError (Column headers are not unique in file *[fname]*)
- RuntimeError (File *[fname]* has no valid data)
- RuntimeError (File *[fname]* is empty)
- RuntimeError (Invalid column specification)
- ValueError (Column *[column_identifier]* not found)

Note: The row where data starts is auto-detected as the first row that has a number (integer of float) in at least one of its columns

---

__eq__(*other*)

Tests object equality. For example:

```
>>> import tempfile, putil.pcsv
>>> with tempfile.NamedTemporaryFile() as fobj:
...     fname = fobj.name
...     putil.pcsv.write(fname, [['a'], [1]], append=False)
...     obj1 = putil.pcsv.CsvFile(fname, dfilter='a')
...     obj2 = putil.pcsv.CsvFile(fname, dfilter='a')
```

```
...
>>> with tempfile.NamedTemporaryFile() as fobj:
...     fname = fobj.name
...     putil.pcsv.write(fname, [['a'], [2]], append=False)
...     obj3 = putil.pcsv.CsvFile(fname, dfilter='a')
...
>>> obj1 == obj2
True
>>> obj1 == obj3
False
>>> 5 == obj3
False
```

__repr__()

Returns a string with the expression needed to re-create the object. For example:

```
>>> import tempfile, putil.pcsv
>>> with tempfile.NamedTemporaryFile() as fobj:
...     fname = fobj.name
...     putil.pcsv.write(fname, [['a'], [1]], append=False)
...     obj1 = putil.pcsv.CsvFile(fname, dfilter='a')
...     exec("obj2="+repr(obj1))
>>> obj1 == obj2
True
>>> repr(obj1)
"putil.pcsv.CsvFile(fname='...', dfilter=['a'])"
```

add_dfilter(*dfilter*)

Adds more row(s) or column(s) to the existing data filter. Duplicate filter values are eliminated

Parameters **dfilter** (*CsvDataFilter*) – Row and/or column filter

Raises

- RuntimeError (Argument ‘dfilter‘ is not valid)
- RuntimeError (Invalid column specification)
- ValueError (Column *[column_identifier]* not found)

cols(*filtered=False*)

Returns the number of data columns

Parameters **filtered** (*boolean*) – Flag that indicates whether the raw (input) data should be used (False) or whether filtered data should be used (True)

Raises RuntimeError (Argument ‘filtered‘ is not valid)

data(*filtered=False*)

Returns (filtered) file data. The returned object is a list, each item is a sub-list corresponding to a row of data; each item in the sub-lists contains data corresponding to a particular column

Parameters **filtered** (*CsvFiltered*) – Filtering type

Return type list

Raises RuntimeError (Argument ‘filtered‘ is not valid)

dsort (*order*)
Sorts rows

Parameters **order** (*CsvColFilter*) – Sort order

Raises

- RuntimeError (Argument 'order' is not valid)
- RuntimeError (Invalid column specification)
- ValueError (Column *[column_identifier]* not found)

header (*filtered=False*)
Returns the data header. When the raw (input) data is used the data header is a list of the comma-separated values file header if the file is loaded with header (each list item is a column header) or a list of column numbers if the file is loaded without header (column zero is the leftmost column). When filtered data is used the data header is the active column filter, if any, otherwise it is the same as the raw (input) data header

Parameters **filtered** (*boolean*) – Flag that indicates whether the raw (input) data should be used (False) or whether filtered data should be used (True)

Return type list of strings or integers

Raises RuntimeError (Argument 'filtered' is not valid)

replace (*rdata*, *filtered=False*)
Replaces data

Parameters

- **rdata** (*list of lists*) – Replacement data
- **filtered** (*CsvFiltered*) – Filtering type

Raises

- RuntimeError (Argument 'filtered' is not valid)
- RuntimeError (Argument 'rdata' is not valid)
- ValueError (Number of columns mismatch between input and replacement data)
- ValueError (Number of rows mismatch between input and replacement data)

reset_dfilter (*ftype=True*)
Reset (clears) the data filter

Parameters **ftype** (*CsvFiltered*) – Filter type

Raises RuntimeError (Argument 'ftype' is not valid)

rows (*filtered=False*)
Returns the number of data rows

Parameters **filtered** (*boolean*) – Flag that indicates whether the raw (input) data should be used (False) or whether filtered data should be used (True)

Raises RuntimeError (Argument 'filtered' is not valid)

write (*fname=None*, *filtered=False*, *header=True*, *append=False*)
Writes (processed) data to a specified comma-separated values (CSV) file

Parameters

- **fname** (*FileName*) – Name of the comma-separated values file to be written. If None the file from which the data originated is overwritten
- **filtered** (*CsvFiltered*) – Filtering type
- **header** (*string, list of strings or boolean*) – If a list, column headers to use in the file. If boolean, flag that indicates whether the input column headers should be written (True) or not (False)
- **append** (*boolean*) – Flag that indicates whether data is added to an existing file (or a new file is created if it does not exist) (True), or whether data overwrites the file contents (if the file exists) or creates a new file if the file does not exists (False)

Raises

- OSError (File *[fname]* could not be created: *[reason]*)
- RuntimeError (Argument 'append' is not valid)
- RuntimeError (Argument 'filtered' is not valid)
- RuntimeError (Argument 'fname' is not valid)
- RuntimeError (Argument 'header' is not valid)
- ValueError (There is no data to save to file)

cfilter
Sets or returns the column filter

Type *CsvColFilter* or None. If None no column filtering is done

Return type *CsvColFilter* or None

Raises (when assigned)

- RuntimeError (Argument 'cfilter' is not valid)
- RuntimeError (Invalid column specification)
- ValueError (Column *[column_identifier]* not found)

dfilter
Sets or returns the data (row and/or column) filter. The first tuple item is the row filter and the second tuple item is the column filter

Type *CsvDataFilter* or None. If None no data filtering is done

Return type *CsvDataFilter* or None

Raises (when assigned)

- RuntimeError (Argument 'dfilter' is not valid)
- RuntimeError (Invalid column specification)
- ValueError (Column *[column_identifier]* not found)

rfilter
Sets or returns the row filter

Type *CsvRowFilter* or None. If None no row filtering is done

Return type *CsvRowFilter* or None

Raises (when assigned)

- RuntimeError (Argument 'rfilter' is not valid)
- RuntimeError (Invalid column specification)
- ValueError (Argument 'rfilter' is empty)
- ValueError (Column *[column_identifier]* not found)

6.12 pcontracts module

This module is a thin wrapper around the PyContracts library that enables customization of the exception type raised and limited customization of the exception message. Additionally, custom contracts specified via *putil.pcontracts.new_contract()* and enforced via *putil.pcontracts.contract()* register exceptions using the *exh module* , which means that the exceptions raised by these contracts can be automatically documented using the *exdoc module*.

The way a contract is specified is identical to the decorator way of specifying a contract with the PyContracts library. By default a `RuntimeError` exception with the message `'Argument `*[argument_name]*` is not valid'` is raised unless a custom contract specifies a different exception (the token `*[argument_name]*` is replaced by the argument name the contract is attached to). For example, the definitions of the custom contracts *putil.ptypes.file_name()* and *putil.ptypes.file_name_exists()* are:

```
@putil.pcontracts.new_contract()
def file_name(obj):
    r"""
    Validates if an object is a legal name for a file
    (i.e. does not have extraneous characters, etc.)

    :param obj: Object
    :type  obj: any

    :raises: RuntimeError (Argument \`*[argument_name]*\` is not
     valid). The token \*[argument_name]\* is replaced by the name
     of the argument the contract is attached to

    :rtype: None
    """
    msg = putil.pcontracts.get_exdesc()
    # Check that argument is a string
    if not isinstance(obj, str):
        raise ValueError(msg)
    # If file exists, argument is a valid file name, otherwise test
    # if file can be created. User may not have permission to
    # write file, but call to os.access should not fail if the file
    # name is correct
    try:
        if not os.path.exists(obj):
            os.access(obj, os.W_OK)
```

```
    except (TypeError, ValueError):
        raise ValueError(msg)
```

```
@putil.pcontracts.new_contract(
    argument_invalid='Argument `*[argument_name]*` is not valid',
    file_not_found=(OSError, 'File *[fname]* could not be found')
)
def file_name_exists(obj):
    r"""
    Validates if an object is a legal name for a file
    (i.e. does not have extraneous characters, etc.) *and* that the
    file exists

    :param obj: Object
    :type  obj: any

    :raises:
     * OSError (File *[fname]* could not be found). The
       token \*[fname]\* is replaced by the *value* of the
       argument the contract is attached to

     * RuntimeError (Argument \`*[argument_name]*\` is not valid).
       The token \*[argument_name]\* is replaced by the name of
       the argument the contract is attached to

    :rtype: None
    """
    exdesc = putil.pcontracts.get_exdesc()
    msg = exdesc['argument_invalid']
    # Check that argument is a string
    if not isinstance(obj, str):
        raise ValueError(msg)
    # Check that file name is valid
    try:
        os.path.exists(obj)
    except (TypeError, ValueError):
        raise ValueError(msg)
    # Check that file exists
    if not os.path.exists(obj):
        msg = exdesc['file_not_found']
        raise ValueError(msg)
```

This is nearly identical to the way custom contracts are defined using the PyContracts library with two exceptions:

1. To avoid repetition and errors, the exception messages defined in the *putil.pcontracts.new_contract()* decorator are available in the contract definition function via *putil.pcontracts.get_exdesc()*.
2. A PyContracts new contract can return False or raise a `ValueError` exception to indicate a contract breach, however a new contract specified via the *putil.pcontracts.new_contract()* decorator *has* to raise a `ValueError` exception to indicate a contract breach.

Exceptions can be specified in a variety of ways and verbosity is minimized by having reasonable defaults (see *putil.pcontracts.new_contract()* for a full description). What follows is a simple usage example of the two contracts shown above and the exceptions they produce:

```
# pcontracts_example_1.py
from __future__ import print_function
import putil.ptypes
```

```
@putil.pcontracts.contract(name='file_name')
def print_if_fname_valid(name):
    """ Sample function 1 """
    print('Valid file name: {0}'.format(name))

@putil.pcontracts.contract(num=int, name='file_name_exists')
def print_if_fname_exists(num, name):
    """ Sample function 2 """
    print('Valid file name: [{0}] {1}'.format(num, name))
```

```
>>> import os
>>> from docs.support.pcontracts_example_1 import *
>>> print_if_fname_valid('some_file.txt')
Valid file name: some_file.txt
>>> print_if_fname_valid('invalid_fname.txt\0')
Traceback (most recent call last):
    ...
RuntimeError: Argument `name` is not valid
>>> fname = os.path.join('..', 'docs', 'pcontracts.rst')
>>> print_if_fname_exists(10, fname)
Valid file name: [10] ...pcontracts.rst
>>> print_if_fname_exists('hello', fname)
Traceback (most recent call last):
    ...
RuntimeError: Argument `num` is not valid
>>> print_if_fname_exists(5, 'another_invalid_fname.txt\0')
Traceback (most recent call last):
    ...
RuntimeError: Argument `name` is not valid
>>> print_if_fname_exists(5, '/dev/null/some_file.txt')
Traceback (most recent call last):
    ...
OSError: File /dev/null/some_file.txt could not be found
```

6.12.1 Functions

putil.pcontracts.**all_disabled**()
: Wraps PyContracts all_disabled() function. From the PyContracts documentation: "Returns true if all contracts are disabled"

putil.pcontracts.**disable_all**()
: Wraps PyContracts disable_all() function. From the PyContracts documentation: "Disables all contract checks"

putil.pcontracts.**enable_all**()
: Wraps PyContracts enable_all() function. From the PyContracts documentation: "Enables all contract checks. Can be overridden by an environment variable"

putil.pcontracts.**get_exdesc**()
: Retrieves the contract exception(s) message(s). If the custom contract is specified with only one exception the return value is the message associated with that exception; if the custom contract is specified with several exceptions, the return value is a dictionary whose keys are the exception names and whose values are the exception messages.

  Raises RuntimeError (Function object could not be found for function *[function_name]*)

  Return type string or dictionary

  For example:

```
    # pcontracts_example_2.py
    import putil.pcontracts

    @putil.pcontracts.new_contract('Only one exception')
    def custom_contract_a(name):
        msg = putil.pcontracts.get_exdesc()
        if not name:
            raise ValueError(msg)

    @putil.pcontracts.new_contract(ex1='Empty name', ex2='Invalid name')
    def custom_contract_b(name):
        msg = putil.pcontracts.get_exdesc()
        if not name:
            raise ValueError(msg['ex1'])
        elif name.find('[') != -1:
            raise ValueError(msg['ex2'])
```

In `custom_contract1()` the variable `msg` contains the string `'Only one exception'`, in `custom_contract2()` the variable `msg` contains the dictionary `{'ex1':'Empty name', 'ex2':'Invalid name'}`.

6.12.2 Decorators

putil.pcontracts.**contract**(**args*, ***kwargs*)

putil.pcontracts.**new_contract**(**args*, ***kwargs*)

6.13 pinspect module

This module supplements Python's introspection capabilities. The class *putil.pinspect.Callables* "traces" modules and produces a database of callables (functions, classes, methods and class properties) and their attributes (callable type, file name, starting line number). Enclosed functions and classes are supported. For example:

```
# pinspect_example_1.py
from __future__ import print_function
import math

def my_func(version):
    """ Enclosing function """
    class MyClass(object):
        """ Enclosed class """
        if version == 2:
            import docs.support.python2_module as pm
        else:
            import docs.support.python3_module as pm

        def __init__(self, value):
            self._value = value

        def _get_value(self):
            return self._value

        value = property(_get_value, pm._set_value, None, 'Value property')
```

```
def print_name(name):
    print('My name is {0}, and sqrt(2) = {1}'.format(name, math.sqrt(2)))
```

with

```
# python2_module.py
def _set_value(self, value):
    self._value = value+2
```

and

```
# python3_module.py
def _set_value(self, value):
    self._value = value+3
```

gives:

```
>>> from __future__ import print_function
>>> import docs.support.pinspect_example_1, putil.pinspect, sys
>>> cobj = putil.pinspect.Callables(
...     [sys.modules['docs.support.pinspect_example_1'].__file__]
... )
>>> print(cobj)
Modules:
   docs.support.pinspect_example_1
Classes:
   docs.support.pinspect_example_1.my_func.MyClass
docs.support.pinspect_example_1.my_func: func (9-25)
docs.support.pinspect_example_1.my_func.MyClass: class (11-25)
docs.support.pinspect_example_1.my_func.MyClass.__init__: meth (18-20)
docs.support.pinspect_example_1.my_func.MyClass._get_value: meth (21-23)
docs.support.pinspect_example_1.my_func.MyClass.value: prop (24-25)
docs.support.pinspect_example_1.print_name: func (26-27)
```

The numbers in parenthesis indicate the line number in which the callable starts and ends within the file it is defined in.

6.13.1 Functions

putil.pinspect.**get_function_args**(*func*, *no_self=False*, *no_varargs=False*)

Returns a tuple of the function argument names in the order they are specified in the function signature

Parameters

- **func** (*function object*) – Function
- **no_self** (*boolean*) – Flag that indicates whether the function argument *self*, if present, is included in the output (False) or not (True)
- **no_varargs** (*boolean*) – Flag that indicates whether keyword arguments are included in the output (True) or not (False)

Return type tuple

For example:

```
>>> import putil.pinspect
>>> class MyClass(object):
...     def __init__(self, value, **kwargs):
...         pass
```

```
    ...
    >>> putil.pinspect.get_function_args(MyClass.__init__)
    ('self', 'value', '**kwargs')
    >>> putil.pinspect.get_function_args(
    ...     MyClass.__init__, no_self=True
    ... )
    ('value', '**kwargs')
    >>> putil.pinspect.get_function_args(
    ...     MyClass.__init__, no_self=True, no_varargs=True
    ... )
    ('value',)
    >>> putil.pinspect.get_function_args(
    ...     MyClass.__init__, no_varargs=True
    ... )
    ('self', 'value')
```

putil.pinspect.**get_module_name**(*module_obj*)

Retrieves the module name from a module object

Parameters **module_obj** (*object*) – Module object

Return type string

Raises

- RuntimeError (Argument ‘module_obj‘ is not valid)
- RuntimeError (Module object ‘*[module_name]*‘ could not be found in loaded modules)

For example:

```
    >>> import putil.pinspect
    >>> putil.pinspect.get_module_name(sys.modules['putil.pinspect'])
    'putil.pinspect'
```

putil.pinspect.**is_object_module**(*obj*)

Tests if the argument is a module object

Parameters **obj** (*any*) – Object

Return type boolean

putil.pinspect.**is_special_method**(*name*)

Tests if a callable name is a special Python method (has a ’__’ prefix and suffix)

Parameters **name** (*string*) – Callable name

Return type boolean

putil.pinspect.**private_props**(*obj*)

Yields private properties of an object. A private property is defined as one that has a single underscore (_) before its name

Parameters **obj** (*object*) – Object

Returns iterator

6.13.2 Classes

class putil.pinspect.**Callables**(*fnames=None*)

Bases: object

Generates a list of module callables (functions, classes, methods and class properties) and gets their attributes (callable type, file name, lines span). Information from multiple modules can be stored in the callables database of the object by repeatedly calling *putil.pinspect.Callables.trace()* with different module file names. A *putil.pinspect.Callables* object retains knowledge of which modules have been traced so repeated calls to *putil.pinspect.Callables.trace()* with the *same* module object will *not* result in module re-traces (and the consequent performance hit)

Parameters **fnames** (*list of strings or None*) – File names of the modules to trace. If None no immediate tracing is done

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument 'fnames' is not valid)

__add__(*other*)

Merges two objects

Raises RuntimeError (Conflicting information between objects)

For example:

```
>>> import putil.eng, putil.exh, putil.pinspect, sys
>>> obj1 = putil.pinspect.Callables(
...     [sys.modules['putil.exh'].__file__]
... )
>>> obj2 = putil.pinspect.Callables(
...     [sys.modules['putil.eng'].__file__]
... )
>>> obj3 = putil.pinspect.Callables([
... sys.modules['putil.exh'].__file__,
... sys.modules['putil.eng'].__file__,
... ])
>>> obj1 == obj3
False
>>> obj1 == obj2
False
>>> obj1+obj2 == obj3
True
```

__copy__()

Copies object. For example:

```
>>> import copy, putil.exh, putil.pinspect, sys
>>> obj1 = putil.pinspect.Callables(
...     [sys.modules['putil.exh'].__file__]
... )
>>> obj2 = copy.copy(obj1)
>>> obj1 == obj2
True
```

__eq__(*other*)

Tests object equality. For example:

```
>>> import putil.eng, putil.exh, putil.pinspect, sys
>>> obj1 = putil.pinspect.Callables(
...     [sys.modules['putil.exh'].__file__]
... )
>>> obj2 = putil.pinspect.Callables(
...     [sys.modules['putil.exh'].__file__]
```

```
... )
>>> obj3 = putil.pinspect.Callables(
...     [sys.modules['putil.eng'].__file__]
... )
>>> obj1 == obj2
True
>>> obj1 == obj3
False
>>> 5 == obj3
False
```

__iadd__(*other*)

Merges an object into an existing object

Raises RuntimeError (Conflicting information between objects)

For example:

```
>>> import putil.eng, putil.exh, putil.pinspect, sys
>>> obj1 = putil.pinspect.Callables(
...     [sys.modules['putil.exh'].__file__]
... )
>>> obj2 = putil.pinspect.Callables(
...     [sys.modules['putil.eng'].__file__]
... )
>>> obj3 = putil.pinspect.Callables([
...     sys.modules['putil.exh'].__file__,
...     sys.modules['putil.eng'].__file__,
... ])
>>> obj1 == obj3
False
>>> obj1 == obj2
False
>>> obj1 += obj2
>>> obj1 == obj3
True
```

__nonzero__()

Returns `False` if no modules have been traced, `True` otherwise. For example:

```
>>> from __future__ import print_function
>>> import putil.eng, putil.pinspect, sys
>>> obj = putil.pinspect.Callables()
>>> if obj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: False
>>> obj.trace([sys.modules['putil.eng'].__file__])
>>> if obj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: True
```

__repr__()

Returns a string with the expression needed to re-create the object. For example:

```
>>> import putil.exh, putil.pinspect, sys
>>> obj1 = putil.pinspect.Callables(
```

```
...     [sys.modules['putil.exh'].__file__]
... )
>>> repr(obj1)
"putil.pinspect.Callables(['...exh.py'])"
>>> exec("obj2="+repr(obj1))
>>> obj1 == obj2
True
```

__str__()
Returns a string with a detailed description of the object's contents. For example:

```
>>> from __future__ import print_function
>>> import putil.pinspect, os, sys
>>> import docs.support.pinspect_example_1
>>> cobj = putil.pinspect.Callables([
...     sys.modules['docs.support.pinspect_example_1'].__file__
... ])
>>> print(cobj)
Modules:
   ...pinspect_example_1
Classes:
   ...pinspect_example_1.my_func.MyClass
...pinspect_example_1.my_func: func (9-25)
...pinspect_example_1.my_func.MyClass: class (11-25)
...pinspect_example_1.my_func.MyClass.__init__: meth (18-20)
...pinspect_example_1.my_func.MyClass._get_value: meth (21-23)
...pinspect_example_1.my_func.MyClass.value: prop (24-25)
...pinspect_example_1.print_name: func (26-27)
```

The numbers in parenthesis indicate the line number in which the callable starts and ends within the file it is defined in

load(*callables_fname*)
Loads traced modules information from a JSON file. The loaded module information is merged with any existing module information

Parameters **callables_fname** (*FileNameExists*) – File name

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument 'callables_fname' is not valid)

refresh()
Re-traces modules which have been modified since the time they were traced

save(*callables_fname*)
Saves traced modules information to a JSON file. If the file exists it is overwritten

Parameters **callables_fname** (*FileName*) – File name

Raises RuntimeError (Argument 'fname' is not valid)

trace(*fnames*)
Generates a list of module callables (functions, classes, methods and class properties) and gets their attributes (callable type, file name, lines span)

Parameters **fnames** (*list*) – File names of the modules to trace

Raises

- OSError (File *[fname]* could not be found)

- RuntimeError (Argument ‘fnames‘ is not valid)

callables_db

Returns the callables database

Return type dictionary

The callable database is a dictionary that has the following structure:

- **full callable name** *(string)* – Dictionary key. Elements in the callable path are separated by periods (`'.'`). For example, method `my_method()` from class `MyClass` from module `my_module` appears as `'my_module.MyClass.my_method'`
  - **callable properties** *(dictionary)* – Dictionary value. The elements of this dictionary are:
  - **type** *(string)* – `'class'` for classes, `'meth'` for methods, `'func'` for functions or `'prop'` for properties or class attributes
  - **code_id** *(tuple or None)* – A tuple with the following items:
    - **file name** *(string)* – the first item contains the file name where the callable can be found
    - **line number** *(integer)* – the second item contains the line number in which the callable code starts (including decorators)
  - **last_lineno** *(integer)* – line number in which the callable code ends (including blank lines and comments regardless of their indentation level)

reverse_callables_db

Returns the reverse callables database

Return type dictionary

The reverse callable database is a dictionary that has the following structure:

- **callable id** *(tuple)* – Dictionary key. Two-element tuple in which the first tuple item is the file name where the callable is defined and the second tuple item is the line number where the callable definition starts
- **full callable name** *(string)* – Dictionary value. Elements in the callable path are separated by periods (`'.'`). For example, method `my_method()` from class `MyClass` from module `my_module` appears as `'my_module.MyClass.my_method'`

6.14 plot module

This module can be used to create high-quality, presentation-ready X-Y graphs quickly and easily

6.14.1 Class hierarchy

The properties of the graph (figure in Matplotlib parlance) are defined in an object of the *putil.plot.Figure* class.

Each figure can have one or more panels, whose properties are defined by objects of the *putil.plot.Panel* class. Panels are arranged vertically in the figure and share the same independent axis. The limits of the independent axis of the figure result from the union of the limits of the independent axis of all the panels. The independent axis is shown by default in the bottom-most panel although it can be configured to be in any panel or panels.

Each panel can have one or more data series, whose properties are defined by objects of the *putil.plot.Series* class. A series can be associated with either the primary or secondary dependent axis of the panel. The limits of the primary and secondary dependent axis of the panel result from the union of the primary and secondary dependent data points of all the series associated with each axis. The primary axis is shown on the left of the panel and the secondary axis is shown on the right of the panel. Axes can be linear or logarithmic.

The data for a series is defined by a source. Two data sources are provided: the *putil.plot.BasicSource* class provides basic data validation and minimum/maximum independent variable range bounding. The *putil.plot.CsvSource* class builds upon the functionality of the *putil.plot.BasicSource* class and offers a simple way of accessing data from a comma-separated values (CSV) file. Other data sources can be programmed by inheriting from the *putil.plot.functions.DataSource* abstract base class (ABC). The custom data source needs to implement the following methods: `__str__`, `_set_indep_var` and `_set_dep_var`. The latter two methods set the contents of the independent variable (an increasing real Numpy vector) and the dependent variable (a real Numpy vector) of the source, respectively.

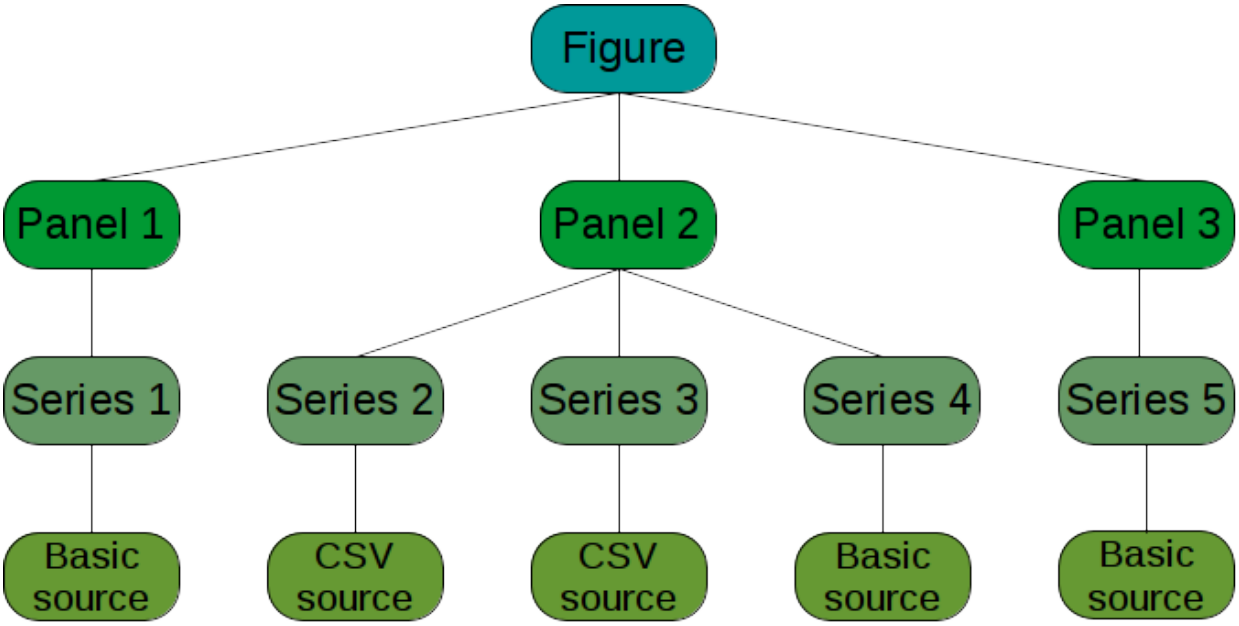

Figure 1: Example diagram of the class hierarchy of a figure. In this particular example the figure consists of 3 panels. Panel 1 has a series whose data comes from a basic source, panel 2 has three series, two of which come from comma-separated values (CSV) files and one that comes from a basic source. Panel 3 has one series whose data comes from a basic source.

6.14.2 Axes tick marks

Axes tick marks are selected so as to create the most readable graph. Two global variables control the actual number of ticks, *putil.plot.constants.MIN_TICKS* and *putil.plot.constants.SUGGESTED_MAX_TICKS*. In general the number of ticks are between these two bounds; one or two more ticks can be present if a data series uses interpolation and the interpolated curve goes above (below) the largest (smallest) data point. Tick spacing is chosen so as to have the most number of data points "on grid". Engineering notation (i.e. 1K = 1000, 1m = 0.001, etc.) is used for the axis tick marks.

6.14.3 Example

```
# plot_example_1.py
from __future__ import print_function
import numpy
import os
import sys
import putil.plot

def main(fname, no_print):
    """
    Example of how to use the putil.plot library
    to generate presentation-quality plots
    """
    ###
    # Series definition (Series class)
    ###
    # Extract data from a comma-separated (csv)
    # file using the CsvSource class
    wdir = os.path.dirname(__file__)
    csv_file = os.path.join(wdir, 'data.csv')
    series1_obj = [putil.plot.Series(
        data_source=putil.plot.CsvSource(
            fname=csv_file,
            rfilter={'value1':1},
            indep_col_label='value2',
            dep_col_label='value3',
            indep_min=None,
            indep_max=None,
            fproc=series1_proc_func,
            fproc_eargs={'xoffset':1e-3}
        ),
        label='Source 1',
        color='k',
        marker='o',
        interp='CUBIC',
        line_style='-',
        secondary_axis=False
    )]
    # Literal data can be used with the BasicSource class
    series2_obj = [putil.plot.Series(
        data_source=putil.plot.BasicSource(
            indep_var=numpy.array([0e-3, 1e-3, 2e-3]),
            dep_var=numpy.array([4, 7, 8]),
        ),
        label='Source 2',
        color='r',
        marker='s',
        interp='STRAIGHT',
        line_style='--',
        secondary_axis=False
    )]
    series3_obj = [putil.plot.Series(
        data_source=putil.plot.BasicSource(
            indep_var=numpy.array([0.5e-3, 1e-3, 1.5e-3]),
            dep_var=numpy.array([10, 9, 6]),
        ),
        label='Source 3',
```

```
        color='b',
        marker='h',
        interp='STRAIGHT',
        line_style='--',
        secondary_axis=True
    )]
    series4_obj = [putil.plot.Series(
        data_source=putil.plot.BasicSource(
            indep_var=numpy.array([0.3e-3, 1.8e-3, 2.5e-3]),
            dep_var=numpy.array([8, 8, 8]),
        ),
        label='Source 4',
        color='g',
        marker='D',
        interp='STRAIGHT',
        line_style=None,
        secondary_axis=True
    )]
    ###
    # Panels definition (Panel class)
    ###
    panel_obj = putil.plot.Panel(
        series=series1_obj+series2_obj+series3_obj+series4_obj,
        primary_axis_label='Primary axis label',
        primary_axis_units='-',
        secondary_axis_label='Secondary axis label',
        secondary_axis_units='W',
        legend_props={'pos':'lower right', 'cols':1}
    )
    ###
    # Figure definition (Figure class)
    ###
    fig_obj = putil.plot.Figure(
        panels=panel_obj,
        indep_var_label='Indep. var.',
        indep_var_units='S',
        log_indep_axis=False,
        fig_width=4*2.25,
        fig_height=3*2.25,
        title='Library putil.plot Example'
    )
    # Save figure
    output_fname = os.path.join(wdir, fname)
    if not no_print:
        print('Saving image to file {0}'.format(output_fname))
    fig_obj.save(output_fname)

def series1_proc_func(indep_var, dep_var, xoffset):
    """ Process data 1 series """
    return (indep_var*1e-3)-xoffset, dep_var
```

Table 6.1: data.csv file

case	value1	value2	value3
0	0	1	3
1	0	2	3
2	1	1	3.5
3	1	2	5.75
4	1	3	10.11
5	1	4	8.88
6	2	1	1
7	2	2	3

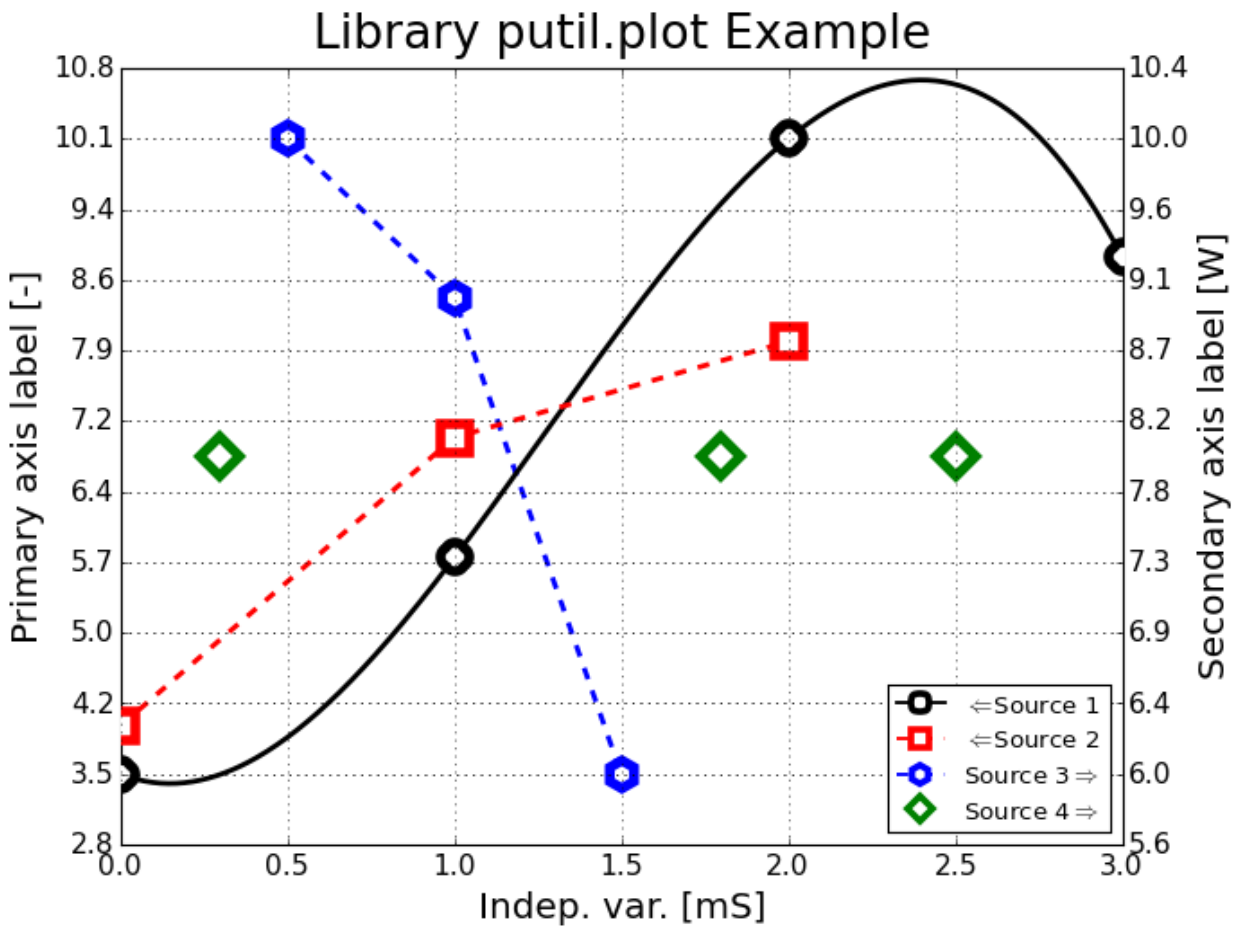

Figure 2: plot_example_1.png generated by plot_example_1.py

6.14.4 Global variables

putil.plot.constants.**AXIS_LABEL_FONT_SIZE = 18**
Axis labels font size in points

Type integer

putil.plot.constants.**LINE_WIDTH = 2.5**
Series line width in points

Type float

putil.plot.constants.**LEGEND_SCALE = 1.5**
Scale factor for panel legend. The legend font size in points is equal to the axis font size divided by the legend scale

Type number

putil.plot.constants.**MARKER_SIZE = 14**
Series marker size in points

Type integer

putil.plot.constants.**MIN_TICKS = 6**
Minimum number of ticks desired for the independent and dependent axis of a panel

Type integer

putil.plot.constants.**PRECISION = 10**
Number of mantissa significant digits used in all computations

Type integer

putil.plot.constants.**SUGGESTED_MAX_TICKS = 10**
Maximum number of ticks desired for the independent and dependent axis of a panel. It is possible for a panel to have more than SUGGESTED_MAX_TICKS in the dependent axis if one or more series are plotted with an interpolation function and at least one interpolated curve goes above or below the maximum and minimum data points of the panel. In this case the panel will have SUGGESTED_MAX_TICKS+1 ticks if some interpolation curve is above the maximum data point of the panel or below the minimum data point of the panel; or the panel will have SUGGESTED_MAX_TICKS+2 ticks if some interpolation curve(s) is(are) above the maximum data point of the panel and below the minimum data point of the panel

Type integer

putil.plot.constants.**TITLE_FONT_SIZE = 24**
Figure title font size in points

Type integer

6.14.5 Functions

putil.plot.**parameterized_color_space**(*param_list*, *offset=0*, *color_space='binary'*)
Computes a color space where lighter colors correspond to lower parameter values

Parameters

- **param_list** (*list*) – Parameter values
- **offset** (*OffsetRange*) – Offset of the first (lightest) color
- **color_space** (*ColorSpaceOption*) – Color palette (case sensitive)

Return type Matplotlib color map

Raises

- RuntimeError (Argument ‘color_space‘ is not valid)
- RuntimeError (Argument ‘offset‘ is not valid)
- RuntimeError (Argument ‘param_list‘ is not valid)
- TypeError (Argument ‘param_list‘ is empty)
- ValueError (Argument ‘color_space‘ is not one of ‘binary’, ‘Blues’, ‘BuGn’, ‘BuPu’, ‘GnBu’, ‘Greens’, ‘Greys’, ‘Oranges’, ‘OrRd’, ‘PuBu’, ‘PuBuGn’, ‘PuRd’, ‘Purples’, ‘RdPu’, ‘Reds’, ‘YlGn’, ‘YlGnBu’, ‘YlOrBr’ or ‘YlOrRd’ (case insensitive))

6.14.6 Classes

class putil.plot.functions.**DataSource**
Bases: object

Abstract base class for data sources. The following example is a minimal implementation of a data source class:

```
# plot_example_2.py
import putil.plot

class MySource(putil.plot.DataSource, object):
    def __init__(self):
        super(MySource, self).__init__()

    def __str__(self):
        return super(MySource, self).__str__()

    def _set_dep_var(self, dep_var):
        super(MySource, self)._set_dep_var(dep_var)

    def _set_indep_var(self, indep_var):
        super(MySource, self)._set_indep_var(indep_var)

    dep_var = property(
        putil.plot.DataSource._get_dep_var, _set_dep_var
    )

    indep_var = property(
        putil.plot.DataSource._get_indep_var, _set_indep_var
    )
```

Warning: The abstract methods listed below need to be defined in a child class

__str__()
Pretty prints the stored independent and dependent variables. For example:

```
>>> from __future__ import print_function
>>> import numpy, docs.support.plot_example_2
>>> obj = docs.support.plot_example_2.MySource()
>>> obj.indep_var = numpy.array([1, 2, 3])
>>> obj.dep_var = numpy.array([-1, 1, -1])
>>> print(obj)
```

```
Independent variable: [ 1.0, 2.0, 3.0 ]
Dependent variable: [ -1.0, 1.0, -1.0 ]
```

_set_dep_var(*dep_var*)
Sets the dependent variable (casting to float type). For example:

```
>>> import numpy, docs.support.plot_example_2
>>> obj = docs.support.plot_example_2.MySource()
>>> obj.dep_var = numpy.array([-1, 1, -1])
>>> obj.dep_var
array([-1.,  1., -1.])
```

_set_indep_var(*indep_var*)
Sets the independent variable (casting to float type). For example:

```
>>> import numpy, docs.support.plot_example_2
>>> obj = docs.support.plot_example_2.MySource()
>>> obj.indep_var = numpy.array([1, 2, 3])
>>> obj.indep_var
array([ 1.,  2.,  3.])
```

class putil.plot.**BasicSource**(*indep_var*, *dep_var*, *indep_min=None*, *indep_max=None*)
Bases: *putil.plot.functions.DataSource*

Objects of this class hold a given data set intended for plotting. It is a convenient way to plot manually-entered data or data coming from a source that does not export to a comma-separated values (CSV) file.

Parameters

- **indep_var** (*IncreasingRealNumpyVector*) – Independent variable vector
- **dep_var** (*RealNumpyVector*) – Dependent variable vector
- **indep_min** (*RealNum or None*) – Minimum independent variable value. If None no minimum thresholding is applied to the data
- **indep_max** (*RealNum or None*) – Maximum independent variable value. If None no maximum thresholding is applied to the data

Return type *putil.plot.BasicSource*

Raises

- RuntimeError (Argument ‘dep_var‘ is not valid)
- RuntimeError (Argument ‘indep_max‘ is not valid)
- RuntimeError (Argument ‘indep_min‘ is not valid)
- RuntimeError (Argument ‘indep_var‘ is not valid)
- ValueError (Argument ‘indep_min‘ is greater than argument ‘indep_max‘)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)
- ValueError (Arguments ‘indep_var‘ and ‘dep_var‘ must have the same number of elements)

__str__()
Prints source information. For example:

```
# plot_example_4.py
import numpy, putil.plot

def create_basic_source():
    obj = putil.plot.BasicSource(
        indep_var=numpy.array([1, 2, 3, 4]),
        dep_var=numpy.array([1, -10, 10, 5]),
        indep_min=2, indep_max=3
    )
    return obj
```

```
>>> from __future__ import print_function
>>> import docs.support.plot_example_4
>>> obj = docs.support.plot_example_4.create_basic_source()
>>> print(obj)
Independent variable minimum: 2
Independent variable maximum: 3
Independent variable: [ 2.0, 3.0 ]
Dependent variable: [ -10.0, 10.0 ]
```

dep_var
Gets or sets the dependent variable data
Type *RealNumpyVector*

Raises (when assigned)
- RuntimeError (Argument ‘dep_var‘ is not valid)
- ValueError (Arguments ‘indep_var‘ and ‘dep_var‘ must have the same number of elements)

indep_max
Gets or sets the maximum independent variable limit. If `None` no maximum thresholding is applied to the data
Type *RealNum* or None

Raises (when assigned)
- RuntimeError (Argument ‘indep_max‘ is not valid)
- ValueError (Argument ‘indep_min‘ is greater than argument ‘indep_max‘)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)

indep_min
Gets or sets the minimum independent variable limit. If `None` no minimum thresholding is applied to the data
Type *RealNum* or None

Raises (when assigned)
- RuntimeError (Argument ‘indep_min‘ is not valid)
- ValueError (Argument ‘indep_min‘ is greater than argument ‘indep_max‘)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)

indep_var
Gets or sets the independent variable data
Type *IncreasingRealNumpyVector*

Raises (when assigned)

- RuntimeError (Argument ‘indep_var‘ is not valid)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)
- ValueError (Arguments ‘indep_var‘ and ‘dep_var‘ must have the same number of elements)

class putil.plot.**CsvSource**(*fname*, *indep_col_label*, *dep_col_label*, *rfilter=None*, *indep_min=None*, *indep_max=None*, *fproc=None*, *fproc_eargs=None*)

Bases: *putil.plot.functions.DataSource*

Objects of this class hold a data set from a CSV file intended for plotting. The raw data from the file can be filtered and a callback function can be used for more general data pre-processing

Parameters

- **fname** (*FileNameExists*) – Comma-separated values file name
- **indep_col_label** (*string*) – Independent variable column label (case insensitive)
- **dep_col_label** (*string*) – Dependent variable column label (case insensitive)
- **rfilter** (*CsvRowFilter or None*) – Row filter specification. If None no row filtering is performed
- **indep_min** (*RealNum or None*) – Minimum independent variable value. If None no minimum thresholding is applied to the data
- **indep_max** (*RealNum or None*) – Maximum independent variable value. If None no maximum thresholding is applied to the data
- **fproc** (*Function or None*) – Data processing function. If None no processing function is used
- **fproc_eargs** (*dictionary or None*) – Data processing function extra arguments. If None no extra arguments are passed to the processing function (if defined)

Return type *putil.plot.CsvSource*

Raises

- OSError (File *[fname]* could not be found)
- RuntimeError (Argument ‘dep_col_label‘ is not valid)
- RuntimeError (Argument ‘dep_var‘ is not valid)
- RuntimeError (Argument ‘fname‘ is not valid)
- RuntimeError (Argument ‘fproc_eargs‘ is not valid)
- RuntimeError (Argument ‘fproc‘ (function *[func_name]*) returned an illegal number of values)
- RuntimeError (Argument ‘fproc‘ is not valid)
- RuntimeError (Argument ‘indep_col_label‘ is not valid)
- RuntimeError (Argument ‘indep_max‘ is not valid)
- RuntimeError (Argument ‘indep_min‘ is not valid)
- RuntimeError (Argument ‘indep_var‘ is not valid)

- RuntimeError (Argument ‘rfilter‘ is not valid)
- RuntimeError (Column headers are not unique in file *[fname]*)
- RuntimeError (File *[fname]* has no valid data)
- RuntimeError (File *[fname]* is empty)
- RuntimeError (Processing function *[func_name]* raised an exception when called with the following arguments: `\n` indep_var: *[indep_var_value]* `\n` dep_var: *[dep_var_value]* `\n` fproc_eargs: *[fproc_eargs_value]* `\n` Exception error: *[exception_error_message]*)
- TypeError (Argument ‘fproc‘ (function *[func_name]*) return value is not valid)
- TypeError (Processed dependent variable is not valid)
- TypeError (Processed independent variable is not valid)
- ValueError (Argument ‘fproc‘ (function *[func_name]*) does not have at least 2 arguments)
- ValueError (Argument ‘indep_min‘ is greater than argument ‘indep_max‘)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)
- ValueError (Argument ‘rfilter‘ is empty)
- ValueError (Arguments ‘indep_var‘ and ‘dep_var‘ must have the same number of elements)
- ValueError (Column *[col_name]* (dependent column label) could not be found in comma-separated file *[fname]* header)
- ValueError (Column *[col_name]* (independent column label) could not be found in comma-separated file *[fname]* header)
- ValueError (Column *[col_name]* in row filter not found in comma- separated file *[fname]* header)
- ValueError (Column *[column_identifier]* not found)
- ValueError (Extra argument ‘*[arg_name]*‘ not found in argument ‘fproc‘ (function *[func_name]*) definition)
- ValueError (Filtered dependent variable is empty)
- ValueError (Filtered independent variable is empty)
- ValueError (Processed dependent variable is empty)
- ValueError (Processed independent and dependent variables are of different length)
- ValueError (Processed independent variable is empty)

__str__()

Prints source information. For example:

```
# plot_example_3.py
import putil.misc, putil.pcsv, sys

def cwrite(fobj, data):
    if sys.hexversion < 0x03000000:
        fobj.write(data)
```

```
            else:
                fobj.write(bytes(data, 'ascii'))

        def write_csv_file(file_handle):
            cwrite(file_handle, 'Col1,Col2\n')
            cwrite(file_handle, '0E-12,10\n')
            cwrite(file_handle, '1E-12,0\n')
            cwrite(file_handle, '2E-12,20\n')
            cwrite(file_handle, '3E-12,-10\n')
            cwrite(file_handle, '4E-12,30\n')

        # indep_var is a Numpy vector, in this example time,
        # in seconds. dep_var is a Numpy vector
        def proc_func1(indep_var, dep_var):
            # Scale time to pico-seconds
            indep_var = indep_var/1e-12
            # Remove offset
            dep_var = dep_var-dep_var[0]
            return indep_var, dep_var

        def create_csv_source():
            with putil.misc.TmpFile(write_csv_file) as fname:
                obj = putil.plot.CsvSource(
                    fname=fname,
                    indep_col_label='Col1',
                    dep_col_label='Col2',
                    indep_min=2E-12,
                    fproc=proc_func1
                )
            return obj
```

```
        >>> from __future__ import print_function
        >>> import docs.support.plot_example_3
        >>> obj = docs.support.plot_example_3.create_csv_source()
        >>> print(obj)
        File name: ...
        Row filter: None
        Independent column label: Col1
        Dependent column label: Col2
        Processing function: proc_func1
        Processing function extra arguments: None
        Independent variable minimum: 2e-12
        Independent variable maximum: +inf
        Independent variable: [ 2.0, 3.0, 4.0 ]
        Dependent variable: [ 0.0, -30.0, 10.0 ]
```

dep_col_label

Gets or sets the dependent variable column label (column name)

Type string

Raises (when assigned)

- RuntimeError (Argument ‘dep_col_label‘ is not valid)
- RuntimeError (Argument ‘fproc‘ (function *[func_name]*) returned an illegal number of values)
- RuntimeError (Processing function *[func_name]* raised an exception when called with the following arguments: \n indep_var: *[indep_var_value]* \n dep_var: *[dep_var_value]* \n fproc_eargs: *[fproc_eargs_value]* \n Exception error: *[ex-*

ception_error_message])
- TypeError (Argument ‘fproc‘ (function *[func_name]*) return value is not valid)
- TypeError (Processed dependent variable is not valid)
- TypeError (Processed independent variable is not valid)
- ValueError (Column *[col_name]* (dependent column label) could not be found in comma-separated file *[fname]* header)
- ValueError (Column *[col_name]* in row filter not found in comma- separated file *[fname]* header)
- ValueError (Filtered dependent variable is empty)
- ValueError (Filtered independent variable is empty)
- ValueError (Processed dependent variable is empty)
- ValueError (Processed independent and dependent variables are of different length)
- ValueError (Processed independent variable is empty)

dep_var

Gets the dependent variable Numpy vector

fname

Gets or sets the comma-separated values file from which data series is to be extracted. It is assumed that the first line of the file contains unique headers for each column

Type string

Raises (when assigned)
- OSError (File *[fname]* could not be found)
- RuntimeError (Argument ‘dep_var‘ is not valid)
- RuntimeError (Argument ‘fname‘ is not valid)
- RuntimeError (Argument ‘fproc‘ (function *[func_name]*) returned an illegal number of values)
- RuntimeError (Argument ‘indep_var‘ is not valid)
- RuntimeError (Argument ‘rfilter‘ is not valid)
- RuntimeError (Column headers are not unique in file *[fname]*)
- RuntimeError (File *[fname]* has no valid data)
- RuntimeError (File *[fname]* is empty)
- RuntimeError (Processing function *[func_name]* raised an exception when called with the following arguments: \n indep_var: *[indep_var_value]* \n dep_var: *[dep_var_value]* \n fproc_eargs: *[fproc_eargs_value]* \n Exception error: *[exception_error_message]*)
- TypeError (Argument ‘fproc‘ (function *[func_name]*) return value is not valid)
- TypeError (Processed dependent variable is not valid)
- TypeError (Processed independent variable is not valid)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)
- ValueError (Argument ‘rfilter‘ is empty)
- ValueError (Arguments ‘indep_var‘ and ‘dep_var‘ must have the same number of elements)
- ValueError (Column *[col_name]* (dependent column label) could not be found in comma-separated file *[fname]* header)
- ValueError (Column *[col_name]* (independent column label) could not be found in comma-separated file *[fname]* header)
- ValueError (Column *[col_name]* in row filter not found in comma- separated file *[fname]* header)
- ValueError (Column *[column_identifier]* not found)
- ValueError (Filtered dependent variable is empty)
- ValueError (Filtered independent variable is empty)

- ValueError (Processed dependent variable is empty)
- ValueError (Processed independent and dependent variables are of different length)
- ValueError (Processed independent variable is empty)

fproc

Gets or sets the data processing function pointer. The processing function is useful for "light" data massaging, like scaling, unit conversion, etc.; it is called after the data has been retrieved from the comma-separated values file and the resulting filtered data set has been bounded (if applicable). If `None` no processing function is used.

When defined the processing function is given two arguments, a Numpy vector containing the independent variable array (first argument) and a Numpy vector containing the dependent variable array (second argument). The expected return value is a two-item Numpy vector tuple, its first item being the processed independent variable array, and the second item being the processed dependent variable array. One valid processing function could be:

```
# indep_var is a Numpy vector, in this example time,
# in seconds. dep_var is a Numpy vector
def proc_func1(indep_var, dep_var):
    # Scale time to pico-seconds
    indep_var = indep_var/1e-12
    # Remove offset
    dep_var = dep_var-dep_var[0]
    return indep_var, dep_var
```

Type *Function* or None

Raises (when assigned)

- RuntimeError (Argument 'fproc' (function *[func_name]*) returned an illegal number of values)
- RuntimeError (Argument 'fproc' is not valid)
- RuntimeError (Processing function *[func_name]* raised an exception when called with the following arguments: `\n` indep_var: *[indep_var_value]* `\n` dep_var: *[dep_var_value]* `\n` fproc_eargs: *[fproc_eargs_value]* `\n` Exception error: *[exception_error_message]*)
- TypeError (Argument 'fproc' (function *[func_name]*) return value is not valid)
- TypeError (Processed dependent variable is not valid)
- TypeError (Processed independent variable is not valid)
- ValueError (Argument 'fproc' (function *[func_name]*) does not have at least 2 arguments)
- ValueError (Extra argument '*[arg_name]*' not found in argument 'fproc' (function *[func_name]*) definition)
- ValueError (Processed dependent variable is empty)
- ValueError (Processed independent and dependent variables are of different length)
- ValueError (Processed independent variable is empty)

fproc_eargs

Gets or sets the extra arguments for the data processing function. The arguments are specified by key-value pairs of a dictionary, for each dictionary element the dictionary key specifies the argument name and the dictionary value specifies the argument value. The extra parameters are passed by keyword so they must appear in the function definition explicitly or keyword variable argument collection must be used (`**kwargs`, for example). If `None` no extra arguments are passed to the processing function (if defined)

Type dictionary or None

Raises (when assigned)

- RuntimeError (Argument ‘fproc_eargs‘ is not valid)
- RuntimeError (Argument ‘fproc‘ (function *[func_name]*) returned an illegal number of values)
- RuntimeError (Processing function *[func_name]* raised an exception when called with the following arguments: \n indep_var: *[indep_var_value]* \n dep_var: *[dep_var_value]* \n fproc_eargs: *[fproc_eargs_value]* \n Exception error: *[exception_error_message]*)
- TypeError (Argument ‘fproc‘ (function *[func_name]*) return value is not valid)
- TypeError (Processed dependent variable is not valid)
- TypeError (Processed independent variable is not valid)
- ValueError (Extra argument ‘*[arg_name]*‘ not found in argument ‘fproc‘ (function *[func_name]*) definition)
- ValueError (Processed dependent variable is empty)
- ValueError (Processed independent and dependent variables are of different length)
- ValueError (Processed independent variable is empty)

For example:

```
# plot_example_5.py
import putil.misc, putil.pcsv, sys
if sys.hexversion < 0x03000000:
    from putil.compat2 import _write
else:
    from putil.compat3 import _write

def write_csv_file(file_handle):
    _write(file_handle, 'Col1,Col2\n')
    _write(file_handle, '0E-12,10\n')
    _write(file_handle, '1E-12,0\n')
    _write(file_handle, '2E-12,20\n')
    _write(file_handle, '3E-12,-10\n')
    _write(file_handle, '4E-12,30\n')

def proc_func2(indep_var, dep_var, par1, par2):
    return (indep_var/1E-12)+(2*par1), dep_var+sum(par2)

def create_csv_source():
    with putil.misc.TmpFile(write_csv_file) as fname:
        obj = putil.plot.CsvSource(
            fname=fname,
            indep_col_label='Col1',
            dep_col_label='Col2',
            fproc=proc_func2,
            fproc_eargs={'par1':5, 'par2':[1, 2, 3]}
        )
    return obj
```

```
>>> from __future__ import print_function
>>> import docs.support.plot_example_5
>>> obj = docs.support.plot_example_5.create_csv_source()
>>> print(obj)
File name: ...
Row filter: None
Independent column label: Col1
Dependent column label: Col2
```

```
    Processing function: proc_func2
    Processing function extra arguments: None
    Independent variable minimum: -inf
    Independent variable maximum: +inf
    Independent variable: [ 10, 11, 12, 13, 14 ]
    Dependent variable: [ 16, 6, 26, -4, 36 ]
```

indep_col_label

Gets or sets the independent variable column label (column name)

Type string

Raises (when assigned)

- RuntimeError (Argument ‘fproc‘ (function *[func_name]*) returned an illegal number of values)
- RuntimeError (Argument ‘indep_col_label‘ is not valid)
- RuntimeError (Processing function *[func_name]* raised an exception when called with the following arguments: `\n` indep_var: *[indep_var_value]* `\n` dep_var: *[dep_var_value]* `\n` fproc_eargs: *[fproc_eargs_value]* `\n` Exception error: *[exception_error_message]*)
- TypeError (Argument ‘fproc‘ (function *[func_name]*) return value is not valid)
- TypeError (Processed dependent variable is not valid)
- TypeError (Processed independent variable is not valid)
- ValueError (Column *[col_name]* (independent column label) could not be found in comma-separated file *[fname]* header)
- ValueError (Column *[col_name]* in row filter not found in comma- separated file *[fname]* header)
- ValueError (Filtered dependent variable is empty)
- ValueError (Filtered independent variable is empty)
- ValueError (Processed dependent variable is empty)
- ValueError (Processed independent and dependent variables are of different length)
- ValueError (Processed independent variable is empty)

indep_max

Gets or sets the maximum independent variable limit. If `None` no maximum thresholding is applied to the data

Type *RealNum* or None

Raises (when assigned)

- RuntimeError (Argument ‘indep_max‘ is not valid)
- ValueError (Argument ‘indep_min‘ is greater than argument ‘indep_max‘)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)

indep_min

Gets or sets the minimum independent variable limit. If `None` no minimum thresholding is applied to the data

Type *RealNum* or None

Raises (when assigned)

- RuntimeError (Argument ‘indep_min‘ is not valid)
- ValueError (Argument ‘indep_min‘ is greater than argument ‘indep_max‘)
- ValueError (Argument ‘indep_var‘ is empty after ‘indep_min‘/‘indep_max‘ range bounding)

indep_var

Gets the independent variable Numpy vector

rfilter

Gets or sets the row filter. If `None` no row filtering is performed

Type *CsvRowFilter* or None

Raises (when assigned)

- RuntimeError (Argument 'fproc' (function *[func_name]*) returned an illegal number of values)
- RuntimeError (Argument 'rfilter' is not valid)
- RuntimeError (Processing function *[func_name]* raised an exception when called with the following arguments: `\n` indep_var: *[indep_var_value]* `\n` dep_var: *[dep_var_value]* `\n` fproc_eargs: *[fproc_eargs_value]* `\n` Exception error: *[exception_error_message]*)
- TypeError (Argument 'fproc' (function *[func_name]*) return value is not valid)
- TypeError (Processed dependent variable is not valid)
- TypeError (Processed independent variable is not valid)
- ValueError (Argument 'rfilter' is empty)
- ValueError (Column *[col_name]* in row filter not found in comma- separated file *[fname]* header)
- ValueError (Filtered dependent variable is empty)
- ValueError (Filtered independent variable is empty)
- ValueError (Processed dependent variable is empty)
- ValueError (Processed independent and dependent variables are of different length)
- ValueError (Processed independent variable is empty)

class `putil.plot.`**Series**(*data_source*, *label*, *color='k'*, *marker='o'*, *interp='CUBIC'*, *line_style='-'*, *secondary_axis=False*)

Bases: `object`

Specifies a series within a panel

Parameters

- **data_source** (*putil.plot.BasicSource*, *putil.plot.CsvSource* or others conforming to the data source specification) – Data source object
- **label** (*string*) – Series label, to be used in the panel legend
- **color** (*polymorphic*) – Series color. All Matplotlib colors are supported
- **marker** (*string or None*) – Marker type. All Matplotlib marker types are supported. None indicates no marker
- **interp** (*InterpolationOption or None*) – Interpolation option (case insensitive), one of None (no interpolation) 'STRAIGHT' (straight line connects data points), 'STEP' (horizontal segments between data points), 'CUBIC' (cubic interpolation between data points) or 'LINREG' (linear regression based on data points). The interpolation option is case insensitive
- **line_style** (*LineStyleOption or None*) – Line style. All Matplotlib line styles are supported. None indicates no line
- **secondary_axis** (*boolean*) – Flag that indicates whether the series belongs to the panel primary axis (False) or secondary axis (True)

Raises

- RuntimeError (Argument 'color' is not valid)
- RuntimeError (Argument 'data_source' does not have an 'dep_var' attribute)

- RuntimeError (Argument ‘data_source‘ does not have an ‘indep_var‘ attribute)
- RuntimeError (Argument ‘data_source‘ is not fully specified)
- RuntimeError (Argument ‘interp‘ is not valid)
- RuntimeError (Argument ‘label‘ is not valid)
- RuntimeError (Argument ‘line_style‘ is not valid)
- RuntimeError (Argument ‘marker‘ is not valid)
- RuntimeError (Argument ‘secondary_axis‘ is not valid)
- TypeError (Invalid color specification)
- ValueError (Argument ‘interp‘ is not one of [’STRAIGHT’, ‘STEP’, ‘CUBIC’, ‘LINREG’] (case insensitive))
- ValueError (Argument ‘line_style‘ is not one of [’-‘, ‘–‘, ‘-.’, ‘:’])
- ValueError (Arguments ‘indep_var‘ and ‘dep_var‘ must have the same number of elements)
- ValueError (At least 4 data points are needed for CUBIC interpolation)

__str__()
Print series object information

color
Gets or sets the series line and marker color. All Matplotlib colors are supported

Type polymorphic

Raises (when assigned)
- RuntimeError (Argument ‘color‘ is not valid)
- TypeError (Invalid color specification)

data_source
Gets or sets the data source object. The independent and dependent data sets are obtained once this attribute is set. To be valid, a data source object must have an `indep_var` attribute that contains a Numpy vector of increasing real numbers and a `dep_var` attribute that contains a Numpy vector of real numbers

Type *putil.plot.BasicSource*, *putil.plot.CsvSource* or others conforming to the data source specification

Raises (when assigned)
- RuntimeError (Argument ‘data_source‘ does not have an ‘dep_var‘ attribute)
- RuntimeError (Argument ‘data_source‘ does not have an ‘indep_var‘ attribute)
- RuntimeError (Argument ‘data_source‘ is not fully specified)
- ValueError (Arguments ‘indep_var‘ and ‘dep_var‘ must have the same number of elements)
- ValueError (At least 4 data points are needed for CUBIC interpolation)

interp
Gets or sets the interpolation option, one of `None` (no interpolation) `'STRAIGHT'` (straight line connects data points), `'STEP'` (horizontal segments between data points), `'CUBIC'` (cubic interpolation between data points) or `'LINREG'` (linear regression based on data points). The interpolation option is case insensitive

Type *InterpolationOption* or None

Raises (when assigned)
- RuntimeError (Argument ‘interp‘ is not valid)

- ValueError (Argument ‘interp‘ is not one of [’STRAIGHT’, ‘STEP’, ‘CUBIC’, ‘LINREG’] (case insensitive))
- ValueError (At least 4 data points are needed for CUBIC interpolation)

label
Gets or sets the series label, to be used in the panel legend if the panel has more than one series
Type string

Raises (when assigned) RuntimeError (Argument ‘label‘ is not valid)

line_style
Sets or gets the line style. All Matplotlib line styles are supported. `None` indicates no line
Type *LineStyleOption*

Raises (when assigned)
- RuntimeError (Argument ‘line_style‘ is not valid)
- ValueError (Argument ‘line_style‘ is not one of [’-‘, ‘–‘, ‘-.’, ‘:’])

marker
Gets or sets the series marker type. All Matplotlib marker types are supported. `None` indicates no marker
Type string or None

Raises (when assigned) RuntimeError (Argument ‘marker‘ is not valid)

secondary_axis
Sets or gets the secondary axis flag; indicates whether the series belongs to the panel primary axis (False) or secondary axis (True)
Type boolean

Raises (when assigned) RuntimeError (Argument ‘secondary_axis‘ is not valid)

class `putil.plot.`**Panel**(*series=None*, *primary_axis_label=’‘*, *primary_axis_units=’‘*, *primary_axis_ticks=None*, *secondary_axis_label=’‘*, *secondary_axis_units=’‘*, *secondary_axis_ticks=None*, *log_dep_axis=False*, *legend_props=None*, *display_indep_axis=False*)
Bases: `object`

Defines a panel within a figure

Parameters

- **series** (*putil.plot.Series or list of putil.plot.Series or None*) – One or more data series
- **primary_axis_label** (*string*) – Primary dependent axis label
- **primary_axis_units** (*string*) – Primary dependent axis units
- **primary_axis_ticks** (*list, Numpy vector or None*) – Primary dependent axis tick marks. If not None overrides automatically generated tick marks if the axis type is linear. If None automatically generated tick marks are used for the primary axis
- **secondary_axis_label** (*string*) – Secondary dependent axis label
- **secondary_axis_units** (*string*) – Secondary dependent axis units
- **secondary_axis_ticks** (*list, Numpy vector or None*) – Secondary dependent axis tick marks. If not None overrides automatically generated tick marks if the axis type is linear. If None automatically generated tick marks are used for the secondary axis

- **log_dep_axis** (*boolean*) – Flag that indicates whether the dependent (primary and /or secondary) axis is linear (False) or logarithmic (True)
- **legend_props** (*dictionary or None*) – Legend properties. See *putil.plot.Panel.legend_props*. If None the legend is placed in the best position in one column
- **display_indep_axis** (*boolean*) – Flag that indicates whether the independent axis is displayed (True) or not (False)

Raises

- RuntimeError (Argument ‘display_indep_axis‘ is not valid)
- RuntimeError (Argument ‘legend_props‘ is not valid)
- RuntimeError (Argument ‘log_dep_axis‘ is not valid)
- RuntimeError (Argument ‘primary_axis_label‘ is not valid)
- RuntimeError (Argument ‘primary_axis_ticks‘ is not valid)
- RuntimeError (Argument ‘primary_axis_units‘ is not valid)
- RuntimeError (Argument ‘secondary_axis_label‘ is not valid)
- RuntimeError (Argument ‘secondary_axis_ticks‘ is not valid)
- RuntimeError (Argument ‘secondary_axis_units‘ is not valid)
- RuntimeError (Argument ‘series‘ is not valid)
- RuntimeError (Legend property ‘cols‘ is not valid)
- RuntimeError (Series item *[number]* is not fully specified)
- TypeError (Legend property ‘pos‘ is not one of [’BEST’, ‘UPPER RIGHT’, ‘UPPER LEFT’, ‘LOWER LEFT’, ‘LOWER RIGHT’, ‘RIGHT’, ‘CENTER LEFT’, ‘CENTER RIGHT’, ‘LOWER CENTER’, ‘UPPER CENTER’, ‘CENTER’] (case insensitive))
- ValueError (Illegal legend property ‘*[prop_name]*‘)
- ValueError (Series item *[number]* cannot be plotted in a logarithmic axis because it contains negative data points)

__bool__()

Returns `True` if the panel has at least a series associated with it, `False` otherwise

Note: This method applies to Python 3.x

__iter__()

Returns an iterator over the series object(s) in the panel. For example:

```
# plot_example_6.py
from __future__ import print_function
import numpy, putil.plot

def panel_iterator_example(no_print):
    source1 = putil.plot.BasicSource(
        indep_var=numpy.array([1, 2, 3, 4]),
        dep_var=numpy.array([1, -10, 10, 5])
    )
```

```
            source2 = putil.plot.BasicSource(
                indep_var=numpy.array([100, 200, 300, 400]),
                dep_var=numpy.array([50, 75, 100, 125])
            )
            series1 = putil.plot.Series(
                data_source=source1,
                label='Goals'
            )
            series2 = putil.plot.Series(
                data_source=source2,
                label='Saves',
                color='b',
                marker=None,
                interp='STRAIGHT',
                line_style='--'
            )
            panel = putil.plot.Panel(
                series=[series1, series2],
                primary_axis_label='Time',
                primary_axis_units='sec',
                display_indep_axis=True
            )
            if not no_print:
                for num, series in enumerate(panel):
                    print('Series {0}:'.format(num+1))
                    print(series)
                    print('')
            else:
                return panel
```

```
        >>> import docs.support.plot_example_6 as mod
        >>> mod.panel_iterator_example(False)
        Series 1:
        Independent variable: [ 1.0, 2.0, 3.0, 4.0 ]
        Dependent variable: [ 1.0, -10.0, 10.0, 5.0 ]
        Label: Goals
        Color: k
        Marker: o
        Interpolation: CUBIC
        Line style: -
        Secondary axis: False

        Series 2:
        Independent variable: [ 100.0, 200.0, 300.0, 400.0 ]
        Dependent variable: [ 50.0, 75.0, 100.0, 125.0 ]
        Label: Saves
        Color: b
        Marker: None
        Interpolation: STRAIGHT
        Line style: --
        Secondary axis: False
```

__nonzero__()

Returns `True` if the panel has at least a series associated with it, `False` otherwise

---

Note: This method applies to Python 2.x

__str__()
Prints panel information. For example:

```
>>> from __future__ import print_function
>>> import docs.support.plot_example_6 as mod
>>> print(mod.panel_iterator_example(True))
Series 0:
   Independent variable: [ 1.0, 2.0, 3.0, 4.0 ]
   Dependent variable: [ 1.0, -10.0, 10.0, 5.0 ]
   Label: Goals
   Color: k
   Marker: o
   Interpolation: CUBIC
   Line style: -
   Secondary axis: False
Series 1:
   Independent variable: [ 100.0, 200.0, 300.0, 400.0 ]
   Dependent variable: [ 50.0, 75.0, 100.0, 125.0 ]
   Label: Saves
   Color: b
   Marker: None
   Interpolation: STRAIGHT
   Line style: --
   Secondary axis: False
Primary axis label: Time
Primary axis units: sec
Secondary axis label: not specified
Secondary axis units: not specified
Logarithmic dependent axis: False
Display independent axis: True
Legend properties:
   cols: 1
   pos: BEST
```

display_indep_axis
Gets or sets the independent axis display flag; indicates whether the independent axis is displayed (True) or not (False)

Type boolean

Raises (when assigned) RuntimeError (Argument ‘display_indep_axis‘ is not valid)

legend_props
Gets or sets the panel legend box properties; this is a dictionary that has properties (dictionary key) and their associated values (dictionary values). Currently supported properties are:

- **pos** (*string*) – legend box position, one of `'BEST'`, `'UPPER RIGHT'`, `'UPPER LEFT'`, `'LOWER LEFT'`, `'LOWER RIGHT'`, `'RIGHT'`, `'CENTER LEFT'`, `'CENTER RIGHT'`, `'LOWER CENTER'`, `'UPPER CENTER'` or `'CENTER'` (case insensitive)
- **cols** (integer) – number of columns of the legend box

If `None` the default used is `{'pos':'BEST', 'cols':1}`

Note: No legend is shown if a panel has only one series in it or if no series has a label

---

Type dictionary

Raises (when assigned)

- RuntimeError (Argument ‘legend_props‘ is not valid)
- RuntimeError (Legend property ‘cols‘ is not valid)

- TypeError (Legend property 'pos' is not one of ['BEST', 'UPPER RIGHT', 'UPPER LEFT', 'LOWER LEFT', 'LOWER RIGHT', 'RIGHT', 'CENTER LEFT', 'CENTER RIGHT', 'LOWER CENTER', 'UPPER CENTER', 'CENTER'] (case insensitive))
- ValueError (Illegal legend property '*[prop_name]*')

log_dep_axis
Gets or sets the panel logarithmic dependent (primary and/or secondary) axis flag; indicates whether the dependent (primary and/or secondary) axis is linear (False) or logarithmic (True)

Type boolean

Raises (when assigned)
- RuntimeError (Argument 'log_dep_axis' is not valid)
- RuntimeError (Argument 'series' is not valid)
- RuntimeError (Series item *[number]* is not fully specified)
- ValueError (Series item *[number]* cannot be plotted in a logarithmic axis because it contains negative data points)

primary_axis_label
Gets or sets the panel primary dependent axis label

Type string

Raises (when assigned) RuntimeError (Argument 'primary_axis_label' is not valid)

primary_axis_scale
Gets the scale of the panel primary axis, `None` if axis has no series associated with it

Type float or None

primary_axis_ticks
Gets the primary axis (scaled) tick locations, `None` if axis has no series associated with it

Type list or None

primary_axis_units
Gets or sets the panel primary dependent axis units

Type string

Raises (when assigned) RuntimeError (Argument 'primary_axis_units' is not valid)

secondary_axis_label
Gets or sets the panel secondary dependent axis label

Type string

Raises (when assigned) RuntimeError (Argument 'secondary_axis_label' is not valid)

secondary_axis_scale
Gets the scale of the panel secondary axis, `None` if axis has no series associated with it

Type float or None

secondary_axis_ticks
Gets the secondary axis (scaled) tick locations, `None` if axis has no series associated with it

Type list or None with it

secondary_axis_units
Gets or sets the panel secondary dependent axis units

Type string

Raises (when assigned) RuntimeError (Argument 'secondary_axis_units' is not valid)

series
Gets or sets the panel series, `None` if there are no series associated with the panel

Type *putil.plot.Series*, list of *putil.plot.Series* or None

Raises (when assigned)

- RuntimeError (Argument ‘series‘ is not valid)
- RuntimeError (Series item *[number]* is not fully specified)
- ValueError (Series item *[number]* cannot be plotted in a logarithmic axis because it contains negative data points)

class putil.plot.**Figure**(*panels=None*, *indep_var_label=’‘*, *indep_var_units=’‘*, *indep_axis_ticks=None*, *fig_width=None*, *fig_height=None*, *title=’‘*, *log_indep_axis=False*)

Bases: object

Generates presentation-quality plots

Parameters

- **panels** (*putil.plot.Panel or list of putil.plot.Panel or None*) – One or more data panels
- **indep_var_label** (*string*) – Independent variable label
- **indep_var_units** (*string*) – Independent variable units
- **indep_axis_ticks** (*list, Numpy vector or None*) – Independent axis tick marks. If not None overrides automatically generated tick marks if the axis type is linear. If None automatically generated tick marks are used for the independent axis
- **fig_width** (*PositiveRealNum* or None) – Hard copy plot width in inches. If None the width is automatically calculated so that there is no horizontal overlap between any two text elements in the figure
- **fig_height** (*PositiveRealNum* or None) – Hard copy plot height in inches. If None the height is automatically calculated so that there is no vertical overlap between any two text elements in the figure
- **title** (*string*) – Plot title
- **log_indep_axis** (*boolean*) – Flag that indicates whether the independent axis is linear (False) or logarithmic (True)

Raises

- RuntimeError (Argument ‘fig_height‘ is not valid)
- RuntimeError (Argument ‘fig_width‘ is not valid)
- RuntimeError (Argument ‘indep_axis_ticks‘ is not valid)
- RuntimeError (Argument ‘indep_var_label‘ is not valid)
- RuntimeError (Argument ‘indep_var_units‘ is not valid)
- RuntimeError (Argument ‘log_indep_axis‘ is not valid)
- RuntimeError (Argument ‘panels‘ is not valid)
- RuntimeError (Argument ‘title‘ is not valid)
- RuntimeError (Figure object is not fully specified)
- RuntimeError (Figure size is too small: minimum width *[min_width]*, minimum height *[min_height]*)
- TypeError (Panel *[panel_num]* is not fully specified)
- ValueError (Figure cannot be plotted with a logarithmic independent axis because panel *[panel_num]*, series *[series_num]* contains negative independent data points)

__bool__()
Returns `True` if the figure has at least a panel associated with it, `False` otherwise

Note: This method applies to Python 3.x

__iter__()
Returns an iterator over the panel object(s) in the figure. For example:

```
# plot_example_7.py
from __future__ import print_function
import numpy, putil.plot

def figure_iterator_example(no_print):
    source1 = putil.plot.BasicSource(
        indep_var=numpy.array([1, 2, 3, 4]),
        dep_var=numpy.array([1, -10, 10, 5])
    )
    source2 = putil.plot.BasicSource(
        indep_var=numpy.array([100, 200, 300, 400]),
        dep_var=numpy.array([50, 75, 100, 125])
    )
    series1 = putil.plot.Series(
        data_source=source1,
        label='Goals'
    )
    series2 = putil.plot.Series(
        data_source=source2,
        label='Saves',
        color='b',
        marker=None,
        interp='STRAIGHT',
        line_style='--'
    )
    panel1 = putil.plot.Panel(
        series=series1,
        primary_axis_label='Average',
        primary_axis_units='A',
        display_indep_axis=False
    )
    panel2 = putil.plot.Panel(
        series=series2,
        primary_axis_label='Standard deviation',
        primary_axis_units=r'$\sqrt{{A}}$',
        display_indep_axis=True
    )
    figure = putil.plot.Figure(
        panels=[panel1, panel2],
        indep_var_label='Time',
        indep_var_units='sec',
        title='Sample Figure'
    )
    if not no_print:
        for num, panel in enumerate(figure):
            print('Panel {0}:'.format(num+1))
            print(panel)
            print('')
    else:
        return figure
```

```
>>> import docs.support.plot_example_7 as mod
>>> mod.figure_iterator_example(False)
Panel 1:
Series 0:
   Independent variable: [ 1.0, 2.0, 3.0, 4.0 ]
   Dependent variable: [ 1.0, -10.0, 10.0, 5.0 ]
   Label: Goals
   Color: k
   Marker: o
   Interpolation: CUBIC
   Line style: -
   Secondary axis: False
Primary axis label: Average
Primary axis units: A
Secondary axis label: not specified
Secondary axis units: not specified
Logarithmic dependent axis: False
Display independent axis: False
Legend properties:
   cols: 1
   pos: BEST

Panel 2:
Series 0:
   Independent variable: [ 100.0, 200.0, 300.0, 400.0 ]
   Dependent variable: [ 50.0, 75.0, 100.0, 125.0 ]
   Label: Saves
   Color: b
   Marker: None
   Interpolation: STRAIGHT
   Line style: --
   Secondary axis: False
Primary axis label: Standard deviation
Primary axis units: $\sqrt{{A}}$
Secondary axis label: not specified
Secondary axis units: not specified
Logarithmic dependent axis: False
Display independent axis: True
Legend properties:
   cols: 1
   pos: BEST
```

__nonzero__()

Returns `True` if the figure has at least a panel associated with it, `False` otherwise

---

Note: This method applies to Python 2.x

__str__()

Prints figure information. For example:

```
>>> from __future__ import print_function
>>> import docs.support.plot_example_7 as mod
>>> print(mod.figure_iterator_example(True))
Panel 0:
   Series 0:
```

```
                Independent variable: [ 1.0, 2.0, 3.0, 4.0 ]
                Dependent variable: [ 1.0, -10.0, 10.0, 5.0 ]
                Label: Goals
                Color: k
                Marker: o
                Interpolation: CUBIC
                Line style: -
                Secondary axis: False
             Primary axis label: Average
             Primary axis units: A
             Secondary axis label: not specified
             Secondary axis units: not specified
             Logarithmic dependent axis: False
             Display independent axis: False
             Legend properties:
                cols: 1
                pos: BEST
          Panel 1:
             Series 0:
                Independent variable: [ 100.0, 200.0, 300.0, 400.0 ]
                Dependent variable: [ 50.0, 75.0, 100.0, 125.0 ]
                Label: Saves
                Color: b
                Marker: None
                Interpolation: STRAIGHT
                Line style: --
                Secondary axis: False
             Primary axis label: Standard deviation
             Primary axis units: $\sqrt{{A}}$
             Secondary axis label: not specified
             Secondary axis units: not specified
             Logarithmic dependent axis: False
             Display independent axis: True
             Legend properties:
                cols: 1
                pos: BEST
          Independent variable label: Time
          Independent variable units: sec
          Logarithmic independent axis: False
          Title: Sample Figure
          Figure width: ...
          Figure height: ...
```

save (*fname*, *ftype='PNG'*)

Saves the figure to a file

Parameters

- **fname** (*FileName*) – File name
- **ftype** (*string*) – File type, either 'PNG' or 'EPS' (case insensitive). The PNG format is a raster format while the EPS format is a vector format

Raises

- RuntimeError (Argument 'fname' is not valid)
- RuntimeError (Argument 'ftype' is not valid)
- RuntimeError (Figure object is not fully specified)
- RuntimeError (Unsupported file type: *[file_type]*)

show()

Displays the figure

Raises
- RuntimeError (Figure object is not fully specified)
- ValueError (Figure cannot be plotted with a logarithmic independent axis because panel *[panel_num]*, series *[series_num]* contains negative independent data points)

axes_list

Gets the Matplotlib figure axes handle list or `None` if figure is not fully specified. Useful if annotations or further customizations to the panel(s) are needed. Each panel has an entry in the list, which is sorted in the order the panels are plotted (top to bottom). Each panel entry is a dictionary containing the following key-value pairs:

- **number** (*integer*) – panel number, panel 0 is the top-most panel
- **primary** (*Matplotlib axis object*) – axis handle for the primary axis, None if the figure has not primary axis
- **secondary** (*Matplotlib axis object*) – axis handle for the secondary axis, None if the figure has no secondary axis

Type list

fig

Gets the Matplotlib figure handle. Useful if annotations or further customizations to the figure are needed. `None` if figure is not fully specified

Type Matplotlib figure handle or None

fig_height

Gets or sets the height (in inches) of the hard copy plot, `None` if figure is not fully specified.

Type *PositiveRealNum* or None

Raises (when assigned) RuntimeError (Argument 'fig_height' is not valid)

fig_width

Gets or sets the width (in inches) of the hard copy plot, `None` if figure is not fully specified

Type *PositiveRealNum* or None

Raises (when assigned) RuntimeError (Argument 'fig_width' is not valid)

indep_axis_scale

Gets the scale of the figure independent axis, `None` if figure is not fully specified

Type float or None if figure has no panels associated with it

indep_axis_ticks

Gets the independent axis (scaled) tick locations, `None` if figure is not fully specified

Type list

indep_var_label

Gets or sets the figure independent variable label

Type string or None

Raises (when assigned)
- RuntimeError (Argument 'indep_var_label' is not valid)
- RuntimeError (Figure object is not fully specified)
- ValueError (Figure cannot be plotted with a logarithmic independent axis because panel *[panel_num]*, series *[series_num]* contains negative independent data points)

indep_var_units

Gets or sets the figure independent variable units

Type string or None

Raises (when assigned)

- RuntimeError (Argument ‘indep_var_units‘ is not valid)
- RuntimeError (Figure object is not fully specified)
- ValueError (Figure cannot be plotted with a logarithmic independent axis because panel *[panel_num]*, series *[series_num]* contains negative independent data points)

log_indep_axis

Gets or sets the figure logarithmic independent axis flag; indicates whether the independent axis is linear (False) or logarithmic (True)

Type boolean

Raises (when assigned)

- RuntimeError (Argument ‘log_indep_axis‘ is not valid)
- RuntimeError (Figure object is not fully specified)
- ValueError (Figure cannot be plotted with a logarithmic independent axis because panel *[panel_num]*, series *[series_num]* contains negative independent data points)

panels

Gets or sets the figure panel(s), `None` if no panels have been specified

Type *putil.plot.Panel*, list of `putil.plot.panel` or None

Raises (when assigned)

- RuntimeError (Argument ‘fig_height‘ is not valid)
- RuntimeError (Argument ‘fig_width‘ is not valid)
- RuntimeError (Argument ‘panels‘ is not valid)
- RuntimeError (Figure object is not fully specified)
- RuntimeError (Figure size is too small: minimum width *[min_width]*, minimum height *[min_height]*)
- TypeError (Panel *[panel_num]* is not fully specified)
- ValueError (Figure cannot be plotted with a logarithmic independent axis because panel *[panel_num]*, series *[series_num]* contains negative independent data points)

title

Gets or sets the figure title

Type string or None

Raises (when assigned)

- RuntimeError (Argument ‘title‘ is not valid)
- RuntimeError (Figure object is not fully specified)
- ValueError (Figure cannot be plotted with a logarithmic independent axis because panel *[panel_num]*, series *[series_num]* contains negative independent data points)

6.15 ptypes module

This module provides several pseudo-type definitions which can be enforced and/or validated with custom contracts defined using the *pcontracts module*

6.15.1 Pseudo-types

ColorSpaceOption

String representing a Matplotlib color space, one `'binary'`, `'Blues'`, `'BuGn'`, `'BuPu'`, `'GnBu'`, `'Greens'`, `'Greys'`, `'Oranges'`, `'OrRd'`, `'PuBu'`, `'PuBuGn'`, `'PuRd'`, `'Purples'`, `'RdPu'`, `'Reds'`, `'YlGn'`, `'YlGnBu'`, `'YlOrBr`, `'YlOrRd'` or `None`

CsvColFilter

String, integer, a list of strings or a list of integers that identify a column or columns within a comma-separated values (CSV) file.

Integers identify a column by position (column 0 is the leftmost column) whereas strings identify the column by name. Columns can be identified either by position or by name when the file has a header (first row of file containing column labels) but only by position when the file does not have a header.

`None` indicates that no column filtering should be done

CsvColSort

Integer, string, dictionary or list of integers, strings or dictionaries that specify the sort direction of a column or columns in a comma-separated values (CSV) file.

The sort direction can be either ascending, specified by the string `'A'`, or descending, specified by the string `'B'` (case insensitive). The default sort direction is ascending.

The column can be specified numerically or with labels depending on whether the CSV file was loaded with or without a header.

The full specification is a dictionary (or list of dictionaries if multiple columns are to be used for sorting) where the key is the column and the value is the sort order, thus valid examples are `{'MyCol':'A'}` and `[{'MyCol':'A'}, {3:'d'}]`.

When the default direction suffices it can be omitted; for example in `[{'MyCol':'D'}, 3]`, the data is sorted first by MyCol in descending order and then by the 4th column (column 0 is the leftmost column in a CSV file) in ascending order

CsvDataFilter

In its most general form a two-item tuple, where one item is of *CsvColFilter* pseudo-type and the other item is of *CsvRowFilter* pseudo-type (the order of the items is not mandated, i.e. the first item could be of pseudo-type CsvRowFilter and the second item could be of pseudo-type CsvColFilter or vice-versa).

The two-item tuple can be reduced to a one-item tuple when only a row or column filter needs to be specified, or simply to an object of either CsvRowFilter or CsvColFilter pseudo-type.

For example, all of the following are valid CsvDataFilter objects: `('MyCol', {'MyCol':2.5})`, `({'MyCol':2.5}, 'MyCol')` (filter in the column labeled MyCol and rows where the column labeled MyCol has the value 2.5), `('MyCol', )` (filter in column labeled MyCol and all rows) and `{'MyCol':2.5}` (filter in all columns and only rows where the column labeled MyCol has the values 2.5)

`None`, `(None, )` or `(None, None)` indicate that no row or column filtering should be done

CsvFiltered

String or a boolean that indicates what type of row and column filtering is to be performed in a comma-separated values (CSV) file. If `True`, `'B'` or `'b'` it indicates that both row- and column-filtering are to be performed; if `False`, `'N'` or `'n'` no filtering is to be performed, if `'R'` or `'r'` only row-filtering is to be performed, if `'C'` or `'c'` only column-filtering is to be performed

CsvRowFilter

Dictionary whose elements are sub-filters with the following structure:

- **column identifier** *(CsvColFilter)* – Dictionary key. Column to filter (as it appears in the comma-separated values file header when a string is given) or column number (when an integer is given, column zero is the leftmost column)
- **value** *(list of strings or numbers, or string or number)* – Dictionary value. Column value to filter if a string or number, column values to filter if a list of strings or numbers

If a row filter sub-filter is a column value all rows which contain the specified value in the specified column are kept for that particular individual filter. The overall data set is the intersection of all the data sets specified by each individual sub-filter. For example, if the file to be processed is:

Ctrl	Ref	Result
1	3	10
1	4	20
2	4	30
2	5	40
3	5	50

Then the filter specification `rfilter = {'Ctrl':2, 'Ref':5}` would result in the following filtered data set:

Ctrl	Ref	Result
2	5	40

However, the filter specification `rfilter = {'Ctrl':2, 'Ref':3}` would result in an empty list because the data set specified by the *Ctrl* individual sub-filter does not overlap with the data set specified by the *Ref* individual sub-filter.

If a row sub-filter is a list, the items of the list represent all the values to be kept for a particular column (strings or numbers). So for example `rfilter = {'Ctrl':[2, 3], 'Ref':5}` would result in the following filtered data set:

Ctrl	Ref	Result
2	5	40
3	5	50

`None` indicates that no row filtering should be done

EngineeringNotationNumber

String with a number represented in engineering notation. Optional leading whitespace can precede the mantissa; optional whitespace can also follow the engineering suffix. An optional sign (+ or -) can precede the mantissa after the leading whitespace. The suffix must be one of `'y'`, `'z'`, `'a'`, `'f'`, `'p'`, `'n'`, `'u'`, `'m'`, `' '` (space), `'k'`, `'M'`, `'G'`, `'T'`, `'P'`, `'E'`, `'Z'` or `'Y'`. The correspondence between suffix and floating point exponent is:

Exponent	Name	Suffix
1E-24	yocto	y
1E-21	zepto	z
1E-18	atto	a
1E-15	femto	f
1E-12	pico	p
1E-9	nano	n
1E-6	micro	u
1E-3	milli	m
1E+0		
1E+3	kilo	k
1E+6	mega	M
1E+9	giga	G
1E+12	tera	T
1E+15	peta	P
1E+18	exa	E
1E+21	zetta	Z
1E+24	yotta	Y

EngineeringNotationSuffix

A single character string, one of `'y'`, `'z'`, `'a'`, `'f'`, `'p'`, `'n'`, `'u'`, `'m'`, `' '` (space), `'k'`, `'M'`, `'G'`, `'T'`, `'P'`, `'E'`, `'Z'` or `'Y'`. *EngineeringNotationNumber* lists the correspondence between suffix and floating point exponent

FileName

String with a valid file name

FileNameExists

String with a file name that exists in the file system

Function

Callable pointer or `None`

IncreasingRealNumpyVector

Numpy vector in which all elements are real (integers and/or floats) and monotonically increasing (each element is strictly greater than the preceding one)

InterpolationOption

String representing an interpolation type, one of `'STRAIGHT'`, `'STEP'`, `'CUBIC'`, `'LINREG'` (case insensitive) or `None`

LineStyleOption

String representing a Matplotlib line style, one of `'-'`, `'--'`, `'-.'`, `':'` or `None`

NodeName

String where hierarchy levels are denoted by node separator characters (`'.'` by default). Node names cannot contain spaces, empty hierarchy levels, start or end with a node separator character.

For this example tree:

root
branch1
|leaf1
|leaf2
branch2

The node names are `'root'`, `'root.branch1'`, `'root.branch1.leaf1'`, `'root.branch1.leaf2'` and `'root.branch2'`

NodesWithData

Dictionary or list of dictionaries; each dictionary must contain exactly two keys:

- **name** (*NodeName*) Node name. See *NodeName* pseudo-type specification
- **data** (*any*) node data

The node data should be an empty list to create a node without data, for example: `{'name':'a.b.c', 'data':[]}`

NonNegativeInteger

Integer greater or equal to zero

OffsetRange

Number in the [0, 1] range

PositiveRealNum

Integer or float greater than zero or `None`

RealNum

Integer, float or `None`

RealNumpyVector

Numpy vector in which all elements are real (integers and/or floats)

6.15.2 Contracts

putil.ptypes.**color_space_option**(*obj*)
Validates if an object is a ColorSpaceOption pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to
- RuntimeError (Argument ‘*[argument_name]*‘ is not one of ‘binary’, ‘Blues’, ‘BuGn’, ‘BuPu’, ‘GnBu’, ‘Greens’, ‘Greys’, ‘Oranges’, ‘OrRd’, ‘PuBu’, ‘PuBuGn’, ‘PuRd’, ‘Purples’, ‘RdPu’, ‘Reds’, ‘YlGn’, ‘YlGnBu’, ‘YlOrBr’ or ‘YlOrRd). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**csv_col_filter**(*obj*)
Validates if an object is a CsvColFilter pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to

Return type None

putil.ptypes.**csv_col_sort**(*obj*)
Validates if an object is a CsvColSort pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to

Return type None

putil.ptypes.**csv_data_filter**(*obj*)
Validates if an object is a CsvDataFilter pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to

Return type None

putil.ptypes.**csv_filtered**(*obj*)
Validates if an object is a CsvFilter pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to

Return type None

putil.ptypes.**csv_row_filter**(*obj*)
Validates if an object is a CsvRowFilter pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to
- ValueError (Argument '*[argument_name]*' is empty). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to

Return type None

putil.ptypes.**engineering_notation_number**(*obj*)
Validates if an object is an EngineeringNotationNumber pseudo-type object

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**engineering_notation_suffix**(*obj*)
Validates if an object is an EngineeringNotationSuffix pseudo-type object

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to

Return type None

putil.ptypes.**non_negative_integer**(*obj*)
Validates if an object is a non-negative (zero or positive) integer

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the *name* of the argument the contract is attached to

Return type None

putil.ptypes.**file_name**(*obj*)
Validates if an object is a legal name for a file (i.e. does not have extraneous characters, etc.)

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**file_name_exists**(*obj*)
Validates if an object is a legal name for a file (i.e. does not have extraneous characters, etc.) *and* that the file exists

Parameters **obj** (*any*) – Object

Raises

- OSError (File *[fname]* could not be found). The token *[fname]* is replaced by the *value* of the argument the contract is attached to
- RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**function**(*obj*)
Validates if an object is a function pointer or `None`

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**increasing_real_numpy_vector**(*obj*)
Validates if an object is IncreasingRealNumpyVector pseudo-type object

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**interpolation_option**(*obj*)
Validates if an object is an InterpolationOption pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to
- RuntimeError (Argument '*[argument_name]*' is not one of ['STRAIGHT', 'STEP', 'CUBIC', 'LINREG'] (case insensitive)). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**line_style_option**(*obj*)
Validates if an object is a LineStyleOption pseudo-type object

Parameters **obj** (*any*) – Object

Raises

- RuntimeError (Argument '*[argument_name]*' is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to
- RuntimeError (Argument '*[argument_name]*' is not one of ['-', '–', '-.', ':']). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**offset_range**(*obj*)
Validates if an object is a number in the [0, 1] range

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**positive_real_num**(*obj*)
Validates if an object is a positive integer, positive float or `None`

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**real_num**(*obj*)
Validates if an object is an integer, float or `None`

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

putil.ptypes.**real_numpy_vector**(*obj*)
Validates if an object is a RealNumpyVector pseudo-type object

Parameters **obj** (*any*) – Object

Raises RuntimeError (Argument ‘*[argument_name]*‘ is not valid). The token *[argument_name]* is replaced by the name of the argument the contract is attached to

Return type None

6.16 test module

This module contains functions to aid in the unit testing of modules in the package (py.test-based)

6.16.1 Functions

6.17 tree module

This module can be used to build, handle, process and search tries

6.17.1 Classes

class putil.tree.**Tree**(*node_separator='.'*)
Bases: `object`

Provides basic trie (radix tree) functionality

Parameters **node_separator** (*string*) – Single character used to separate nodes in the tree

Return type *putil.tree.Tree* object

Raises RuntimeError (Argument ‘node_separator‘ is not valid)

__nonzero__()
Returns `False` if tree object has no nodes, `True` otherwise. For example:

```
>>> from __future__ import print_function
>>> import putil.tree
>>> tobj = putil.tree.Tree()
>>> if tobj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: False
>>> tobj.add_nodes([{'name':'root.branch1', 'data':5}])
>>> if tobj:
...     print('Boolean test returned: True')
... else:
...     print('Boolean test returned: False')
Boolean test returned: True
```

__str__()
Returns a string with the tree 'pretty printed' as a character-based structure. Only node names are shown, nodes with data are marked with an asterisk (*). For example:

```
>>> from __future__ import print_function
>>> import putil.tree
>>> tobj = putil.tree.Tree()
>>> tobj.add_nodes([
...     {'name':'root.branch1', 'data':5},
...     {'name':'root.branch2', 'data':[]},
...     {'name':'root.branch1.leaf1', 'data':[]},
...     {'name':'root.branch1.leaf2', 'data':'Hello world!'}
... ])
>>> print(tobj)
root
branch1 (*)
|leaf1
|leaf2 (*)
branch2
```

Return type Unicode string

add_nodes(*nodes*)
Adds nodes to tree

Parameters **nodes** (*NodesWithData*) – Node(s) to add with associated data. If there are several list items in the argument with the same node name the resulting node data is a list with items corresponding to the data of each entry in the argument with the same node name, in their order of appearance, in addition to any existing node data if the node is already present in the tree

Raises

- RuntimeError (Argument 'nodes' is not valid)
- ValueError (Illegal node name: *[node_name]*)

For example:

```
# tree_example.py
import putil.tree
```

```
        def create_tree():
            tobj = putil.tree.Tree()
            tobj.add_nodes([
                {'name':'root.branch1', 'data':5},
                {'name':'root.branch1', 'data':7},
                {'name':'root.branch2', 'data':[]},
                {'name':'root.branch1.leaf1', 'data':[]},
                {'name':'root.branch1.leaf1.subleaf1', 'data':333},
                {'name':'root.branch1.leaf2', 'data':'Hello world!'},
                {'name':'root.branch1.leaf2.subleaf2', 'data':[]},
            ])
            return tobj
```

```
        >>> from __future__ import print_function
        >>> import docs.support.tree_example
        >>> tobj = docs.support.tree_example.create_tree()
        >>> print(tobj)
        root
        branch1 (*)
        |leaf1
        ||subleaf1 (*)
        |leaf2 (*)
        | subleaf2
        branch2

        >>> tobj.get_data('root.branch1')
        [5, 7]
```

collapse_subtree(*name*, *recursive=True*)

Collapses a sub-tree; nodes that have a single child and no data are combined with their child as a single tree node

Parameters

- **name** (*NodeName*) – Root of the sub-tree to collapse
- **recursive** (*boolean*) – Flag that indicates whether the collapse operation is performed on the whole sub-tree (True) or whether it stops upon reaching the first node where the collapsing condition is not satisfied (False)

Raises

- RuntimeError (Argument ‘name‘ is not valid)
- RuntimeError (Argument ‘recursive‘ is not valid)
- RuntimeError (Node *[name]* not in tree)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
        >>> from __future__ import print_function
        >>> import docs.support.tree_example
        >>> tobj = docs.support.tree_example.create_tree()
        >>> print(tobj)
        root
        branch1 (*)
        |leaf1
        ||subleaf1 (*)
        |leaf2 (*)
```

```
| subleaf2
branch2
>>> tobj.collapse_subtree('root.branch1')
>>> print(tobj)
root
branch1 (*)
|leaf1.subleaf1 (*)
|leaf2 (*)
| subleaf2
branch2
```

`root.branch1.leaf1` is collapsed because it only has one child (`root.branch1.leaf1.subleaf1`) and no data; `root.branch1.leaf2` is not collapsed because although it has one child (`root.branch1.leaf2.subleaf2`) and this child does have data associated with it, `'Hello world!'`

copy_subtree(*source_node*, *dest_node*)

Copies a sub-tree from one sub-node to another. Data is added if some nodes of the source sub-tree exist in the destination sub-tree

Parameters

- **source_name** (*NodeName*) – Root node of the sub-tree to copy from
- **dest_name** (*NodeName*) – Root node of the sub-tree to copy to

Raises

- RuntimeError (Argument ‘dest_node‘ is not valid)
- RuntimeError (Argument ‘source_node‘ is not valid)
- RuntimeError (Illegal root in destination node)
- RuntimeError (Node *[source_node]* not in tree)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
>>> from __future__ import print_function
>>> import docs.support.tree_example
>>> tobj = docs.support.tree_example.create_tree()
>>> print(tobj)
root
branch1 (*)
|leaf1
||subleaf1 (*)
|leaf2 (*)
| subleaf2
branch2
>>> tobj.copy_subtree('root.branch1', 'root.branch3')
>>> print(tobj)
root
branch1 (*)
|leaf1
||subleaf1 (*)
|leaf2 (*)
| subleaf2
branch2
branch3 (*)
 leaf1
 |subleaf1 (*)
```

```
leaf2 (*)
 subleaf2
```

delete_prefix(*name*)

Deletes hierarchy levels from all nodes in the tree

Parameters **nodes** (*NodeName*) – Prefix to delete

Raises

- RuntimeError (Argument 'name' is not a valid prefix)
- RuntimeError (Argument 'name' is not valid)

For example:

```
>>> from __future__ import print_function
>>> import putil.tree
>>> tobj = putil.tree.Tree('/')
>>> tobj.add_nodes([
...     {'name':'hello/world/root', 'data':[]},
...     {'name':'hello/world/root/anode', 'data':7},
...     {'name':'hello/world/root/bnode', 'data':8},
...     {'name':'hello/world/root/cnode', 'data':False},
...     {'name':'hello/world/root/bnode/anode', 'data':['a', 'b']},
...     {'name':'hello/world/root/cnode/anode/leaf', 'data':True}
... ])
>>> tobj.collapse_subtree('hello', recursive=False)
>>> print(tobj)
hello/world/root
anode (*)
bnode (*)
|anode (*)
cnode (*)
 anode
  leaf (*)
>>> tobj.delete_prefix('hello/world')
>>> print(tobj)
root
anode (*)
bnode (*)
|anode (*)
cnode (*)
 anode
  leaf (*)
```

delete_subtree(*nodes*)

Deletes nodes (and their sub-trees) from the tree

Parameters **nodes** (*NodeName* or list of *NodeName*) – Node(s) to delete

Raises

- RuntimeError (Argument 'nodes' is not valid)
- RuntimeError (Node *[node_name]* not in tree)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
>>> from __future__ import print_function
>>> import docs.support.tree_example
>>> tobj = docs.support.tree_example.create_tree()
>>> print(tobj)
root
branch1 (*)
|leaf1
||subleaf1 (*)
|leaf2 (*)
| subleaf2
branch2
>>> tobj.delete_subtree(['root.branch1.leaf1', 'root.branch2'])
>>> print(tobj)
root
branch1 (*)
 leaf2 (*)
  subleaf2
```

flatten_subtree(*name*)

Flattens sub-tree; nodes that have children and no data are merged with each child

Parameters **name** (*NodeName*) – Ending hierarchy node whose sub-trees are to be flattened

Raises

- RuntimeError (Argument ‘name‘ is not valid)
- RuntimeError (Node *[name]* not in tree)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
>>> from __future__ import print_function
>>> import docs.support.tree_example
>>> tobj = docs.support.tree_example.create_tree()
>>> tobj.add_nodes([
...     {'name':'root.branch1.leaf1.subleaf2', 'data':[]},
...     {'name':'root.branch2.leaf1', 'data':'loren ipsum'},
...     {'name':'root.branch2.leaf1.another_subleaf1', 'data':[]},
...     {'name':'root.branch2.leaf1.another_subleaf2', 'data':[]}
... ])
>>> print(str(tobj))
root
branch1 (*)
|leaf1
||subleaf1 (*)
||subleaf2
|leaf2 (*)
| subleaf2
branch2
 leaf1 (*)
  another_subleaf1
  another_subleaf2
>>> tobj.flatten_subtree('root.branch1.leaf1')
>>> print(str(tobj))
root
branch1 (*)
|leaf1.subleaf1 (*)
|leaf1.subleaf2
|leaf2 (*)
```

```
        | subleaf2
        branch2
         leaf1 (*)
          another_subleaf1
          another_subleaf2
        >>> tobj.flatten_subtree('root.branch2.leaf1')
        >>> print(str(tobj))
        root
        branch1 (*)
        |leaf1.subleaf1 (*)
        |leaf1.subleaf2
        |leaf2 (*)
        | subleaf2
        branch2
         leaf1 (*)
          another_subleaf1
          another_subleaf2
```

get_children (*name*)
Gets the children node names of a node

Parameters **name** (*NodeName*) – Parent node name

Return type list of *NodeName*

Raises

- RuntimeError (Argument ‘name‘ is not valid)
- RuntimeError (Node *[name]* not in tree)

get_data (*name*)
Gets the data associated with a node

Parameters **name** (*NodeName*) – Node name

Return type any type or list of objects of any type

Raises

- RuntimeError (Argument ‘name‘ is not valid)
- RuntimeError (Node *[name]* not in tree)

get_leafs (*name*)
Gets the sub-tree leaf node(s)

Parameters **name** (*NodeName*) – Sub-tree root node name

Return type list of *NodeName*

Raises

- RuntimeError (Argument ‘name‘ is not valid)
- RuntimeError (Node *[name]* not in tree)

get_node (*name*)
Gets a tree node structure. The structure is a dictionary with the following keys:

•**parent** (*NodeName*) Parent node name, `''` if the node is the root node

•**children** (*list of NodeName*) Children node names, an empty list if node is a leaf

•**data** (*list*) Node data, an empty list if node contains no data

Parameters **name** (*string*) – Node name

Return type dictionary

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

get_node_children(*name*)

Gets the list of children structures of a node. See *putil.tree.Tree.get_node()* for details about the structure

Parameters **name** (*NodeName*) – Parent node name

Return type list

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

get_node_parent(*name*)

Gets the parent structure of a node. See *putil.tree.Tree.get_node()* for details about the structure

Parameters **name** (*NodeName*) – Child node name

Return type dictionary

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

get_subtree(*name*)

Gets all node names in a sub-tree

Parameters **name** (*NodeName*) – Sub-tree root node name

Return type list of *NodeName*

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
>>> from __future__ import print_function
>>> import docs.support.tree_example, pprint
>>> tobj = docs.support.tree_example.create_tree()
>>> print(tobj)
root
```

```
    branch1 (*)
    |leaf1
    ||subleaf1 (*)
    |leaf2 (*)
    | subleaf2
    branch2
    >>> pprint.pprint(tobj.get_subtree('root.branch1'))
    ['root.branch1',
     'root.branch1.leaf1',
     'root.branch1.leaf1.subleaf1',
     'root.branch1.leaf2',
     'root.branch1.leaf2.subleaf2']
```

is_root(*name*)

Tests if a node is the root node (node with no ancestors)

Parameters **name** (*NodeName*) – Node name

Return type boolean

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

in_tree(*name*)

Tests if a node is in the tree

Parameters **name** (*NodeName*) – Node name to search for

Return type boolean

Raises RuntimeError (Argument 'name' is not valid)

is_leaf(*name*)

Tests if a node is a leaf node (node with no children)

Parameters **name** (*NodeName*) – Node name

Return type boolean

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

make_root(*name*)

Makes a sub-node the root node of the tree. All nodes not belonging to the sub-tree are deleted

Parameters **name** (*NodeName*) – New root node name

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
>>> from __future__ import print_function
>>> import docs.support.tree_example
>>> tobj = docs.support.tree_example.create_tree()
>>> print(tobj)
root
branch1 (*)
|leaf1
||subleaf1 (*)
|leaf2 (*)
| subleaf2
branch2
>>> tobj.make_root('root.branch1')
>>> print(tobj)
root.branch1 (*)
leaf1
|subleaf1 (*)
leaf2 (*)
 subleaf2
```

print_node(*name*)

Prints node information (parent, children and data)

Parameters **name** (*NodeName*) – Node name

Raises

- RuntimeError (Argument 'name' is not valid)
- RuntimeError (Node *[name]* not in tree)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
>>> from __future__ import print_function
>>> import docs.support.tree_example
>>> tobj = docs.support.tree_example.create_tree()
>>> print(tobj)
root
branch1 (*)
|leaf1
||subleaf1 (*)
|leaf2 (*)
| subleaf2
branch2
>>> print(tobj.print_node('root.branch1'))
Name: root.branch1
Parent: root
Children: leaf1, leaf2
Data: [5, 7]
```

rename_node(*name*, *new_name*)

Renames a tree node. It is typical to have a root node name with more than one hierarchy level after using *putil.tree.Tree.make_root()*. In this instance the root node *can* be renamed as long as the new root name has the same or less hierarchy levels as the existing root name

Parameters **name** (*NodeName*) – Node name to rename

Raises

- RuntimeError (Argument 'name' is not valid)

- RuntimeError (Argument 'new_name' has an illegal root node)
- RuntimeError (Argument 'new_name' is an illegal root node name)
- RuntimeError (Argument 'new_name' is not valid)
- RuntimeError (Node *[name]* not in tree)
- RuntimeError (Node *[new_name]* already exists)

Using the same example tree created in *putil.tree.Tree.add_nodes()*:

```
>>> from __future__ import print_function
>>> import docs.support.tree_example
>>> tobj = docs.support.tree_example.create_tree()
>>> print(tobj)
root
branch1 (*)
|leaf1
||subleaf1 (*)
|leaf2 (*)
| subleaf2
branch2
>>> tobj.rename_node(
...     'root.branch1.leaf1',
...     'root.branch1.mapleleaf1'
... )
>>> print(tobj)
root
branch1 (*)
|leaf2 (*)
||subleaf2
|mapleleaf1
| subleaf1 (*)
branch2
```

search_tree(*name*)

Searches tree for all nodes with a specific name

Parameters **name** (*NodeName*) – Node name to search for

Raises RuntimeError (Argument 'name' is not valid)

For example:

```
>>> from __future__ import print_function
>>> import pprint, putil.tree
>>> tobj = putil.tree.Tree('/')
>>> tobj.add_nodes([
...     {'name':'root', 'data':[]},
...     {'name':'root/anode', 'data':7},
...     {'name':'root/bnode', 'data':[]},
...     {'name':'root/cnode', 'data':[]},
...     {'name':'root/bnode/anode', 'data':['a', 'b', 'c']},
...     {'name':'root/cnode/anode/leaf', 'data':True}
... ])
>>> print(tobj)
root
anode (*)
bnode
```

```
    |anode (*)
    cnode
     anode
      leaf (*)
    >>> pprint.pprint(tobj.search_tree('anode'), width=40)
    ['root/anode',
     'root/bnode/anode',
     'root/cnode/anode',
     'root/cnode/anode/leaf']
```

nodes
Gets the name of all tree nodes, `None` if the tree is empty

Return type list of *NodeName* or None

node_separator
Gets the node separator character

Return type string

root_name
Gets the tree root node name, `None` if the *putil.tree.Tree* object has no nodes

Return type *NodeName* or None

root_node
Gets the tree root node structure or `None` if *putil.tree.Tree* object has no nodes. See *putil.tree.Tree.get_node()* for details about returned dictionary

Return type dictionary or None

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

W