brennivin Documentation

Release 0.1.0

Rob Galanakis

August 20, 2014

Brennivin is a series of utility libraries that were developed while working on EVE Online, by CCP Games. It includes all sorts of generally useful Python functionality that is useful in almost any domain. Rather than reinvent the wheel, or look all over the place for the right screw driver, Brennivin was created to have a single well tested and documented place for utilities.

This is an Open Source fork of that internal code. Less useful or Windows-specific portions have been stripped, and may be added as companion packages in the future.

Anyway, there's no sort of framework here, just cohesive modules that will save you some work. Hopefully Brennivin has what you need.

- **brennivin** on GitHub: https://github.com/rgalanakis/brennivin
- **brennivin** on Read the Docs: http://brennivin.readthedocs.org/
- **brennivin** on Travis-CI: https://travis-ci.org/rgalanakis/brennivin
- **brennivin** on Coveralls: https://coveralls.io/r/rgalanakis/brennivin

CHAPTER 1

Brennivin's modules

Brennivin includes a number of useful utility modules that augment the Python libraries with similar names. Others are just plain handy. Here's a rundown of what's included:

- `brennivin.dochelpers` provides functions for creating prettier documentation,
- `brennivin.ioutils` provides retry and timeout decorators,
- `brennivin.itertoolsext` provides functions for working with iterables, like `first`, `last`, and all sorts of other useful things (it's probably the most useful module in here).
- `brennivin.logutils` has helpful formatters, and functions for working with and cleaning up timestamped log files,
- `brennivin.nicenum` provides pretty number and memory formatting,
- `brennivin.osutils` has functions that should be on `os` and `os.path` in the first place, such as a recursive file lister, and context manager to change the cwd, and more.
- `brennivin.platformutils` has a few functions to tell you about the process and OS/hardware,
- `brennivin.preferences` provides an object that will serialize preferences,
- `brennivin.pyobjcomparer` allows the comparison of complex Python objects, such as when you need to compare floats with a tolerance, but those floats are in deeply nested data structures,
- `brennivin.testhelpers` has gobs of useful assertion methods (numbers, sequences, folders, files), and other nice helpers,
- `brennivin.threadutils` has a not-crap Thread class that can raise exceptions instead of swalling them, a restartable timer, and mechanisms for communication such as tokens and signals.
- `brennivin.traceback2` is like the `traceback` module but will include locals, and has more controls for formatting,
- `brennivin.yamlext` contains some helpers for simplifying yaml reading/writing,
- and `brennivin.zipfileutils` has useful functions for creating zip files that you wish were on the `zipfile` module.

There's a lot more useful functionality in each of these modules than what's listed above, so check them out!

And in a neater form:

1.1 brennivin.dochelpers module

Some helpers for creating pretty documentation. Usually only of use with `brennivin` but you can use for your own purposes if you want.

You can use the `default` or `unsupplied` objects to act as a unique sentinel, and `ignore` for “private” parameters (such as those with a leading underscore).

1.1.1 Members

`brennivin.dochelpers.`**`named_none`**(*s*)
Return an instance where `bool(obj) == False` and will return `s` when stringified.

`brennivin.dochelpers.`**`pretty_func`**(*func*, *reprstr*)
Returns a callable that has the same behavior as `func`, and will return `reprstr` when strigified.

`brennivin.dochelpers.`**`pretty_module_func`**(*func*)
Same as `pretty_func()` but for module functions (will automatically pull the repr string from its module and name).

`brennivin.dochelpers.`**`pretty_value`**(*value*, *reprstr*)
Returns as a callable that will return `value` when invoked with no parameters, and will return `reprstr` when stringified.

1.2 brennivin.ioutils module

Contains utilities for working with IO, such as the `retry` and `timeout` decorators, and the `is_local_port_open()` function.

Also defines `Timeout` which is used in IO-heavy areas of brennivin.

1.2.1 Members

`brennivin.ioutils.`**`is_local_port_open`**(*port*)
Returns True if `port` is open on the local host. Note that this only checks whether the port is open at an instant in time and may be bound after this function checks but before it even returns!

class `brennivin.ioutils.`**`retry`**(*attempts=2*, *excfilter=(<type ‘exceptions.Exception’>,)*, *wait=0*, *backoff=1*, *sleepfunc=None*)
Decorator used for retrying an operation multiple times. After each retry, the wait will be multiplied by backoff.

Parameters

- **attempts** – Number of attemts to retry, total. Must be >= 1.
- **excfilter** – Types of exceptions to catch when an attempt fails.
- **wait** – Initial amount of time to sleep before retrying. Must be >= 0. If 0, do not sleep between retries.
- **backoff** – Multiplier for time to sleep, multiplied by number of attempts. Must be >= 1.
- **sleepfunc** – The function used to sleep between retries. Default to `time.sleep()`.

class `brennivin.ioutils.`**`timeout`**`(`*timeoutSecs=5*`)`

Decorator used for aborting operations after they have timed out. Raises a `Timeout` on timeout.

The actual work happens on a new thread. Patch the timeout.start_thread method with your own compatible method if you do not want to use a thread to run the operation, such as if you want to use tasklets or greenlets from stackless or gevent.

1.3 brennivin.itertoolsext module

Like the `itertools` module but contains even more stuff! Also puts `itertools` namespace into this module so you can safely just use `brennivin.itertoolsext` instead of `itertools`.

Most useful are the sequence functions based off the LINQ extension methods in C#.

Contains the `Bundle` and `FrozenDict` utility types.

Contains functions for walking trees depth and breadth first. While a depth-first recursive function is easy to write and you should feel free to write your own instead of using `treeyield_depthfirst()`, a breadth-first is much more difficult and you should consider using `treeyield_breadthfirst()`.

Also contains functions for getting/setting attrs and keys of compound objects and dicts, such as `set_compound_attr()` and `get_compound_item()`.

1.3.1 Members

class `brennivin.itertoolsext.`**`Bundle`**`(`*seq=None*, ***kwargs*`)`

Allowed .notation access of dict keys.

Note that any keys that would otherwise hide a dict attribute will be ignored.

```
>>> Bundle({'spam': 'eggs'}).spam
'eggs'
```

class `brennivin.itertoolsext.`**`FrozenDict`**

An immutable version of a dict.

Raises `TypeError` for any unsupported (mutable) operations.

`brennivin.itertoolsext.`**`all`**`(`*seq*, *predicate=<type 'bool'>*`)`

Returns True if all items in seq return True for predicate(item).

`brennivin.itertoolsext.`**`any`**`(`*seq*, *predicate=<type 'bool'>*`)`

Returns True if any item in seq returns True for predicate(item).

`brennivin.itertoolsext.`**`bucket`**`(`*seq*, *keyprojection=lambda x: x*, *valueprojection=lambda x: x*`)`

Buckets items in seq according to their keyprojection. Returns a dict where the keys are all unique values from keyprojections, and the values are lists containing the valueprojection of all items in that bucket.

```
>>> seq = ('a', 1), ('b', 2), ('a', 3)
>>> bucket(seq, lambda x: x[0], lambda x: x[1])
{'a': [1, 3], 'b': [2]}
```

`brennivin.itertoolsext.`**`count`**`(`*seq*, *predicate=None*`)`

Return the number of items in seq. If predicate is specified, return the number of items in seq that predicate is true for.

`brennivin.itertoolsext.`**`datespan`**`(`*startdate*, *enddate*, *delta=datetime.timedelta(1)*`)`

Yield datetimes while walking from startdate to enddate. If startdate is later than endddate, will count backwards starting at startdate.

Note that startdate is inclusive (will always be the first yield), enddate is exclusive (it or later will never be yielded).

Parameters

- **startdate** – The first date to yield.
- **enddate** – The date to stop yielding at.
- **delta** – The size of the step.

```
>>> dt = _datetime.datetime
>>> len(list(datespan(dt(2010, 2, 20), dt(2010, 4, 21))))
60
>>> len(list(datespan(dt(2010, 2, 20), dt(2010, 4, 21),
...                    _datetime.timedelta(days=2))))
30
>>> len(list(datespan(dt(2010, 4, 21), dt(2010, 2, 20))))
60
```

`brennivin.itertoolsext.`**`del_compound_item`**`(`*collection*, **indices*`)`

Delete element in collection that `indices` point to.

`brennivin.itertoolsext.`**`dict_add`**`(`*alpha*, *beta*, *adder_function=None*`)`

Adds any keys and valued from the second given dictionary to the first one using the given adder function. Any keys in the beta dict not present in alpha will be added there as they are in beta. If alpha and beta share keys, the adder function is applied to their values. If no adder function is supplied a simple numerical addition is used.

An adder function should take in two parameters, the value from alpha and the value from beta in those cases where they share keys, and should return the desired value after addition.

Example:

```
>>> summary = {'done': 38, 'errors': 0}
>>> current = {'done': 4, 'warnings': 3}
>>> dict_add(summary, current)
>>> print summary
{'errors': 0, 'done': 42, 'warnings': 3}
```

Parameters

- **alpha** (*dict*) – The dict to add stuff to
- **beta** (*dict*) – The dict supplying stuff

`brennivin.itertoolsext.`**`first`**`(`*seq*, *predicate=None*`)`

Return the first item in seq. If predicate is specified, return the first item in seq that predicate returns True for. Raises StopIteration if no item is found.

`brennivin.itertoolsext.`**`first_or_default`**`(`*seq*, *predicate=None*, *default=None*`)`

Return the first item in seq. If predicate is specified, return the first item in seq that predicate returns True for. Returns `default` if sequence is empty or no item matches criteria.

`brennivin.itertoolsext.`**`flatmap`**`(`*function*, *sequence*`)`

Return a generator that yields the equivalent of: `chain(map(function, sequence))`.

Parameters

- **function** – Function that takes an item in sequence and returns a sequence.

- **sequence** – Any iterable object.

`brennivin.itertoolsext.`**`get_compound_attr`**`(`*obj*, **namesandindices*`)`

Like `getattr`, but takes a sequence of names and indices to get compound attr from `obj`.

`get_compound_attr(x, ['y', 0, 'z']` is equivalent to `x.y[0].z`.

`brennivin.itertoolsext.`**`get_compound_item`**`(`*collection*, **indices*`)`

Return the value of the element in collection that the indicies point to.

E.g.:

```
>>> get_compound_item(['a', {'b': 'value'}], 1, 'b')
'value'
```

`brennivin.itertoolsext.`**`groupby2`**`(`*seq*, *keyfunc*`)`

Groups a sequence by a key selector. Yields 2-item tuples of (key, list of items that share key).

Parameters

- **seq** – Sequence of items. Items should be collections or other complex items. Ie, it doesn't make sense to group a list of integers, and such a sequence will give unpredictable results.
- **keyfunc** – Callable that takes an item in seq and returns the key for grouping.

This is a more intuitive version of itertools.groupby (which is used internally) since it will sort the data for you, and has a more sensible return signature.

```
>>> seq = ('a', 1), ('a', 2), ('b', 3)
>>> dict(groupby2(seq, lambda pair: pair[0]))
{'a': [('a', 1), ('a', 2)], 'b': [('b', 3)]}
```

`brennivin.itertoolsext.`**`groups_of_n`**`(`*seq*, *n*`)`

Return list of groups. A group is `seq` split at size `n`.

```
>>> groups_of_n([1,2,3,4,5,6], 3)
[[1, 2, 3], [4, 5, 6]]
```

`brennivin.itertoolsext.`**`last`**`(`*seq*, *predicate=None*`)`

Return the last item in seq. If predicate is specified, return last item in seq that predicate returns True for.

Raises StopIteration If seq is empty or no item matches predicate criteria.

`brennivin.itertoolsext.`**`last_or_default`**`(`*seq*, *predicate=None*, *default=None*`)`

Return the last item in seq. If predicate is specified, return the last item in seq that predicate returns true for. Return `default` if seq is empty or no item matches predicate criteria.

`brennivin.itertoolsext.`**`set_compound_attr`**`(`*obj*, *value*, **namesandindices*`)`

Like `setattr`, but takes a sequence of names and indices to set compound attr on `obj`.

`set_compound_attr(x, ['y', 0, 'z'], v)` is equivalent to `x.y[0].z = v`.

`brennivin.itertoolsext.`**`set_compound_item`**`(`*collection*, *value*, **indices*`)`

Mutates element of collection that the indicies point to, setting it to value.

E.g.:

```
>>> coll = ['a', {'b': None}]
>>> set_compound_item(coll, 'value', 1, 'b')
>>> print coll
['a', {'b': 'value'}]
```

`brennivin.itertoolsext.`**`shuffle`**(*col*, *maxattempts=5*)

Return a new shuffled collection and ensure it is shuffled. A regular random.shuffle could return the same as the input for small sequences.

Asserts after more than maxattempts at shuffle have been made.

`brennivin.itertoolsext.`**`single`**(*seq*)

Returns the only item in seq. If seq is empty or has more than one item, raise StopIteration.

`brennivin.itertoolsext.`**`skip`**(*sequence*, *number*)

Returns a generator that skips `number` of items in `sequence`. Similar to `sequence[number:]`.

`brennivin.itertoolsext.`**`take`**(*seq*, *number*, *predicate=None*)

Returns a list with len <= number from items in seq.

`brennivin.itertoolsext.`**`treeyield_breadthfirst`**(*node*, *getchildren*, *getchild=None*, *yieldnode=False*)

Yields a tree in breadth first order.

Parameters

- **node** – The node to start at.
- **getchildren** – A function to return the children of 'node'.
- **getchild** – If provided, getChildren should return an int (the index of the child in 'node'). This function will be called with (node, index).
- **yieldnode** – If True, yield 'node' argument.

`brennivin.itertoolsext.`**`treeyield_depthfirst`**(*node*, *getchildren*, *getchild=None*, *yieldnode=False*)

Yields a tree in depth first order.

Parameters

- **node** – The node to start at.
- **getchildren** – A function to return the children of 'node'.
- **getchild** – If provided, getChildren should return an int (the index of the child in 'node'). This function will be called with (node, index).
- **yieldnode** – If True, yield 'node' argument.

`brennivin.itertoolsext.`**`unique`**(*seq*, *transform=lambda x: x*)

Returns an iterator of unique elements in seq. If transform is provided, will apply the method to items in seq as the keys for uniqueness.

1.4 brennivin.logutils module

Useful stuff for working with the stdlib's `logging` module.

See the `Fmt` class for commonly used format strings/formatters.

If you need a `NullHandler`, use the one from this package to add Python 2.6 compatibility.

There are various functions for working with timestamped log filenames (getting the timestamped name, cleaning up old versions, etc).

Use `get_filenames_from_loggers()` to get all the logging filenames currently registered.

1.4.1 Members

class `brennivin.logutils.`**`Fmt`**

Commonly used format strings and formatters. Formatter instances begin with *'FMT'*.

Attributes should be strings of letters that describe the format:

- N: name
- T: asctime
- L: levelname
- M: message

class `brennivin.logutils.`**`MultiLineIndentFormatter`**(*fmt=None*, *datefmt=None*, *sep=' '*)

Indents every newline character in a formatted logrecord to have the same indentation as the formatted record's header.

`brennivin.logutils.`**`get_filenames_from_loggers`**(*loggers=None*, *_loggingmodule=None*)

Get the filenames of all log files from loggers. If not supplied, use all loggers from `logging` module.

`brennivin.logutils.`**`get_timestamped_logfilename`**(*folder*, *basename=None*, *ext='.log'*, *fmt='%Y-%m-%d-%H-%M-%S'*, *timestruct=None*, *_getpid=<built-in function getpid>*)

Using default keyword arguments return filename `<folder>/<basename>_<timestamp>_<pid>.log` in the app's folder in ccptechart prefs.

Parameters

- **folder** – Folder to put file into.
- **basename** – The prefix of the log filename. If None, use `os.path.basename(folder)`.

`brennivin.logutils.`**`remove_old_files`**(*root*, *namepattern='*'*, *maxfiles=1*)

Removes the oldest files that match `namePattern` inside of `rootDir`, so that only `maxfiles` of those matches remain.

Parameters **maxfiles** – Number of files to keep. If 0, remove all files.

`brennivin.logutils.`**`timestamp`**(*fmt*, *timestruct=None*)

Return timestamp by calling `time.strftime(fmt, timestruct())`.

Parameters

- **fmt** – format str, see http://docs.python.org/2/library/datetime.html?highlight=time.strftime#strftime-strptime-behavior for details
- **timestruct** – 9-tuple, see `time.gmtime()` for details.

`brennivin.logutils.`**`timestamped_filename`**(*filename*, *fmt='%Y-%m-%d-%H-%M-%S'*, *timestruct=None*, *sep='_'*)

Given a filename, return a new filename '{head}_{formatted timestruct}.{ext}'.

Parameters

- **filename** – The filename.
- **fmt** – The format string.
- **timestruct** – A named tuple instance such as from time.localtime(). Defaults to time.gmtime().
- **sep** – Separator between the filename and the time.

```
>>> timestamped_filename(r'C:lah.log', timestruct=(2010,9,8,7,6,5,4,3,0))
r'C:lah_2010-09-08-07-06-05.log'
```

brennivin.logutils.**wrap_line**(*s*, *maxlines*, *maxlen=254*, *pfx='- '*)

Parameters

- **s** – input string
- **maxlines** – max amount of lines, or 0 for no limit
- **maxlen** – max length of any one line
- **pfx** – prefix for lines after first (counts towards line length)

1.5 brennivin.nicenum module

Functions for formatting numbers in a pretty way.

1.5.1 Members

brennivin.nicenum.**format**(*num*, *precision*)

Returns a string representation for a floating point number that is rounded to the given precision and displayed with commas and spaces.

```
>>> print format(123567.0, 1000)
124,000
>>> print format(5.3918e-07, 1e-10)
0.000 000 539 2
```

This kind of thing is wonderful for producing tables for human consumption.

brennivin.nicenum.**format_memory**(*val*)

Pretty formatting of memory.

1.6 brennivin.osutils module

Functionality that you wish was on `os` and `os.path`. And some that you don't!

1.6.1 Members

brennivin.osutils.**abspathex**(*path*, *relative_to*, *_ignore_this=False*)

Returns a normalized absoluted version of the pathname `path`, relative to `relativeTo` directory. `relativeTo` must be an actual directory.

Parameters

- **path** – The relative path to make absolute.
- **relative_to** – The filename to make path absolute to. This path will be made absolute before using it.
- **_ignore_this** – For internal use only.

NOTE: This actually works by changing the cwd temporarily, but will always set it back. That said, there could be some side effects so use with care.

`brennivin.osutils.`**`change_cwd`**(**args*, ***kwds*)
Context manager for temporarily changing the cwd.

Parameters **cwd** – The directory to use as the cwd.

`brennivin.osutils.`**`change_environ`**(**args*, ***kwds*)
Context manager for temporarily changing an os.environ entry. A `newvalue` of None will delete the entry.

`brennivin.osutils.`**`change_ext`**(*path*, *ext*)
Changes the extension of path to be ext. If path has no extension, ext will be appended.

Parameters

- **path** – The path.
- **ext** – The extension, should begin in a '.'.

`brennivin.osutils.`**`copy`**(*src*, *dst*)
Copies src to dst, recursively making directories for dst if they do not exist.

src and dst should be filenames (not directories).

`brennivin.osutils.`**`crc_from_filename`**(*filename*)
Returns the 32-bit crc for the file at filename.

`brennivin.osutils.`**`iter_files`**(*directory*, *pattern='*'*)
Returns a generator of files under directory that match pattern.

`brennivin.osutils.`**`listdirex`**(*path*, *pattern='*.*'*)
Return absolute filepaths in `path` that matches `pattern`.

`brennivin.osutils.`**`makedirs`**(*path*, *mode=511*)
Like `os.makedirs`, but will not fail if directory exists.

`brennivin.osutils.`**`mktemp`**(**args*, ***kwargs*)
Returns an absolute temporary filename. A replacement for python's deprecated mktemp. Will call mkstemp and close the resultant file.

Args are the same as tempfile.mkstemp

`brennivin.osutils.`**`path_components`**(*path*)
Return list of a path's components.

`brennivin.osutils.`**`purename`**(*filename*)
Returns the basename of a path without the extension.

`brennivin.osutils.`**`split3`**(*path*)
Returns a tuple of (dirname, filename without ext, ext).

1.7 brennivin.platformutils module

Functionality for learning about the current platform/executable.

Supports finding the Python flavor (ExeFile, Maya, 26, 27), whether the OS is 64 bit Windows, and whether the current process is 64 bits.

1.7.1 Members

`brennivin.platformutils.`**`cpu_count`**`(`*_multiprocmod=<IGNORE>*`)`
Number of virtual or physical CPUs on this system.

`brennivin.platformutils.`**`get_interpreter_flavor`**`(`*_exepath=<IGNORE>*, *_vinfo=<IGNORE>*`)`
Return one of the `'EXE'`-prefixed consts showing which interpreter is in use.

`brennivin.platformutils.`**`is_64bit_process`**`(`*_structmod=<IGNORE>*`)`
Return True if the current process is 64 bits, False if 32, otherwise raises OSError.

`brennivin.platformutils.`**`is_64bit_windows`**`()`
Return true if the current OS is a 64 bit windows OS, False if not. Behavior unreliable on other OSes.

1.8 brennivin.preferences module

Contains the `Preferences` class for serializing preferences. It is very simple (prefs are a dict of dicts) and flexible (can be serialized using user-provided functions).

1.8.1 Members

class `brennivin.preferences.`**`Preferences`**`(`*filename*, *onloaderror=None*`)`
Handles the serialization of preferences values. Pickled makes the following guarantees about its data:

- If the file or directories does not exist, they will be created.
- If the file exists but is corrupt (either the data is corrupt, or it is filled with not-a-dict), preferences will be reset.

Parameters

- **filename** – The filename where preferences will be saved. Directories will be created automatically on init if they do not exist.
- **onloaderror** – If provided, invoke this function in the case of an error on Load. If None, just log out that an error occured. Errors on Save will still be raised, of course.

Override the `loader()`, `dumper()`, and `openmode()` methods to use a serializaer other than json.

`dumper``(`*obj*, *fp*`)`
Like `json.dump(obj, fp)`

`get``(`*region*, *variable*, *defaultValue*`)`
Get a preference value from the pickled data.

Parameters

- **region** – The parent group that owns the variable.
- **variable** – The name of the stored variable.
- **defaultValue** – Value to return if the key or region do not exist.

`load``()`
Load a pickled file into the local prefs dict. If the data in the file is not a dict, reset all prefs to be a dict.

`loader``(`*fp*`)`
Like `json.load(fp)`

openmode()
‘t’ or ‘b’ indicating the way to open the persisted file.

save()
Save the internal data in a pickle file.

set(*region*, *variable*, *value*)
Register a value to be stored in a cPickle file.

Parameters

- **region** – The parent group that owns the variable.
- **variable** – The name of the variable.
- **value** – The value to be stored as region.variable.

setdefault(*region*, *variable*, *defaultValue*)
If `get()` `(region, variable)` is not set, performs a `set()` `(region, variable, defaultValue)` and returns `defaultValue`.

1.9 brennivin.pyobjcomparer module

Module for comparing arbitrary python object structures using the `compare()` function.

Uses a map of type to comparison method to perform comparisons recursively.

This module can be pretty easily refactored so that the comparison methods, tolerance, etc., are customizable.

This module is necessary so we can customize the built-in behavior of python’s structure comparer, which very almost suits our needs, except for comparison of floats and similar.

1.9.1 Members

`brennivin.pyobjcomparer.`**assert_compare**(*a*, *b*, *print_objs=True*)
Raise AssertionError if `a` != `b`, else return None.

`brennivin.pyobjcomparer.`**compare**(*a*, *b*)
Returns `True` if `a` equals `b` using the special sauce logic, `False` if not.

Return type bool

`brennivin.pyobjcomparer.`**get_compound_diff**(*a*, *b*)
If `a != b` return list of compound elements from `a` that leads to the differing value.

If `a == b` return empty list.

```
>>> a = [{'first': [{'second': 0}]}]
>>> b = [{'first': [{'second': 1}]}]
>>> get_compound_diff(a, b)
[0, 'first', 0, 'second']
```

An edge-case exists where `a != b`, but the point of inequality occurs immediately, without traversing into the input structures (e.g. if one or both inputs are not lists or dicts). In this case the returned list becomes `[a, b]`.

```
>>> get_compound_diff(object(), 'foo')
[<object object at 0x...>, 'foo']
```

Return type list

1.10 brennivin.testhelpers module

Auxilliary classes to facilitate testing.

1.10.1 Members

`unittest`: Points to the `unittest` module in Python >= 2.7, and `unittest2` in Python <= 2.6. This aids in creating version-agnostic test cases.

class `brennivin.testhelpers.`**`CallCounter`**(*call*)

Counts the number of times the instance is called. Available via the `count` attribute.

Generally you should not create this class directly, and use the `no_params` and `all_params` class methods. Use the former when calling this should take no arguments, and the latter when it should take any arguments.

class `brennivin.testhelpers.`**`FakeTestCase`**

Sometimes, you want to use one of the assertion methods in this module, but don't have a testcase. You can just use an instance of this.

class `brennivin.testhelpers.`**`Patcher`**(*obj*, *attrname*, *newvalue=DEFAULT*)

Context manager that stores `getattr(obj, attrname)`, sets it to `newvalue` on enter, and restores it on exit.

Parameters **newvalue** – If _undefined, use a new Mock.

`brennivin.testhelpers.`**`assertBetween`**(*tc*, *a*, *b*, *c*, *eq=False*)

Asserts that:

```
if eq:
    a <= b <= c
else:
    a < b < c
```

`brennivin.testhelpers.`**`assertCrcEqual`**(*testcase*, *calcpath*, *idealpath*, *asLib=False*)

Asserts if crcs of paths are not equal. If `DIFF_FILES_ON_CRC_FAIL` is True, launch P4MERGE to diff the files.

Parameters **asLib** – If True, do not print or show diff (function is being used as part of another assert function and not as a standalone).

`brennivin.testhelpers.`**`assertEqualPretty`**(*testcase*, *calculated*, *ideal*, *msg=None*)

Prints ideal and calculated on two lines, for easier analysis of what's different. Useful for sequences.

Parameters

- **testcase** – An instance of unittest.TestCase
- **ideal** – The value that should have been calculated.
- **calculated** – the value that was calculated.

`brennivin.testhelpers.`**`assertFoldersEqual`**(*testcase*, *calcfolder*, *idealfolder*, *compare=assertCrcEqual*)

Asserts if any differences are found between two folders.

Parameters **compare** – Assertion method to use. Pass in a custom function that switches off of extensions if you want.

`brennivin.testhelpers.`**`assertJsonEqual`**(*calc*, *ideal*, *out=<open file '<stderr>', mode 'w' at 0x7f21ca5d51e0>*)
Asserts if `calc != ideal`. Will print the diff between the json dump of `calc` and `ideal` to `out` before asserting, as a debugging aid.

`brennivin.testhelpers.`**`assertNumberSequencesEqual`**(*testcase*, *a*, *b*, *tolerance=0*)
Assert that for an element in sequence `a` the corresponding element in `b` is equal within `tolerance`.

Also assert if the two sequences are not the same length.

`brennivin.testhelpers.`**`assertNumbersEqual`**(*testcase*, *a*, *b*, *tolerance=0*, *msg=None*)
Asserts if the sizes are not within `tolerance`.

`brennivin.testhelpers.`**`assertPermissionbitsEqual`**(*testcase*, *calcpath*, *idealpath*, *bitgetter=os.stat(<path>)[0]*)
Asserts if permission bits are not equal.

`brennivin.testhelpers.`**`assertTextFilesEqual`**(*testcase*, *calcpath*, *idealpath*, *compareLines=None*)
Asserts if the files are not equal. It first compares crcs, and if that fails, it compares the file contents as text (ie, mode 'r' and not 'rb'). The reason for this is that there can be discrepancies between newlines.

Parameters **compareLines** – Callable that takes (calculated line, ideal line) and should assert if they are not equal. Defaults to `testcase.assertEqual(calc line, ideal line)`.

`brennivin.testhelpers.`**`assertXmlEqual`**(*a*, *b*)
Asserts two xml documents are equal.

`brennivin.testhelpers.`**`compareXml`**(*x1*, *x2*, *reporter=lambda x: x*)
Compares two xml elements. If they are equal, return True. If not, return False. Differences are reported by calling the `reporter` parameter, such as

```
reporter('Tags do not match: Foo and Bar')
```

`brennivin.testhelpers.`**`patch`**(*testcase*, **args*)
Create and enter `Patcher` that will be exited on `testcase` teardown.

Parameters **args** – Passed to the `Patcher`.

Returns The monkey patcher's newvalue.

1.11 brennivin.threadutils module

Things to make working with threading in Python easier. Note, you should generally avoid using threads, but sometimes you need them! Check out `brennivin.uthread` for a tasklet based solution.

Contains useful Thread subclasses:

- `ExceptionalThread`, which brings proper exceptions to threads.
- `NotAThread`, which is useful for mocking threads because it runs synchronously when `start()` is called.
- `TimerExt`: A cancellable/restartable `threading.Timer`.

Some useful threading-related utilities:

- `ChunkIter`, useful for chunking work on a background thread and reporting it to another thread in chunks (useful for UI).
- `memoize()`, a caching decorator that can be threadsafe (vital if you want a singleton that has some expensive global state to construct, for example). There is also `expiring_memoize` for a time-based solution.
- `token`, a simple threading token that can be set/queried, useful for inter-thread communication.
- `Signal`, used for registering and signaling events in a process.
- `join_timeout()`, raises an error if a thread is alive after a join.

1.11.1 Members

class `brennivin.threadutils.`**`ChunkIter`**(*iterable_*, *callback*, *chunksize=50*)

Object that can be used to iterate over a collection on a background thread and report progress in a callback. This is useful when iteration of items is slow (such as if it is an expensive map or filter) and can be asynchronous.

Iteration starts as soon as the object is created.

Parameters

- **iterable** – An iterable object, such as a list or generator. If the iterable is to be mapped and filtered, use `itertools.imap` and `itertools.ifilter` to pass in generators that perform the mapping and filtering, so it too can be done on the background thread.
- **callback** – A callable that takes a list of items as yielded by `iterable_`.
- **chunksize** – Chunks will be reported back to `callable` with lists of `chunksize` items (the last chunk will be leftovers).

If you do not want to use threading, override or patch the `start_thread` class method to use whatever library.

`Cancel`()

Call to cancel the iteration. Not be instantaneous.

`IsFinished`()

Returns True if the iteration is finished.

`WaitChunks`(*chunks=1*, *sleep_interval=1*)

Waits for `chunks` amount of chunks to be reported. Useful directly after initialization, to wait for some seed of items to be iterated.

Parameters

- **chunks** – Number of chunks to wait for.
- **sleep_interval** – Amount of time to sleep before checking to see if new chunks are reported.

`cancel`()

Call to cancel the iteration. Not be instantaneous.

`is_finished`()

Returns True if the iteration is finished.

`wait_chunks`(*chunks=1*, *sleep_interval=1*)

Waits for `chunks` amount of chunks to be reported. Useful directly after initialization, to wait for some seed of items to be iterated.

Parameters

- **chunks** – Number of chunks to wait for.

- **sleep_interval** – Amount of time to sleep before checking to see if new chunks are reported.

wait_for_completion(*timeout=None*)
`threading.Thread.join(timeout)()` on the background thread.

class `brennivin.threadutils.`**ExceptionalThread**(**args*, ***kwargs*)
Drop-in subclass for a regular `threading.Thread`.

If an error occurs during `run()`:

- Sets `self.exc_info = sys.exc_info()`
- Calls `self.excepted.Fire(self.exc_info)`
- If `sys.excepthook` is not the default, invoke it with self.exc_info. The default just writes to stderr, so no point using it.
- If `self.reraise` is True, reraise the original error when all this handling is complete.

If an error occured, it will also be raised on `join()`.

Parameters kwargs – Same as Thread, except with a `reraise` key (default None). If reraise is True, invoke the Thread exception handling after ExceptionalThread's exception handling. If False, do not invoke Thread's exception handling. If None, only invoke Thread's exception handling if `excepted` has no delegates and sys.excepthook is the default.

If you are joining on the thread at any point, you should always set reraise to False, since join will reraise any exceptions on the calling thread.

There's usually little point using True because Thread's exception handling because it just writes to stderr.

class `brennivin.threadutils.`**NotAThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, *verbose=None*)
A thread whose `start()` method runs synchronously. Useful to say `ExceptionalThread = NotAThread` if you want to debug a program without threading.

class `brennivin.threadutils.`**Signal**(*eventdoc=None*, *onerror=traceback.print_exception*)
Maintains a collection of delegates that can be easily fired.

Listeners can add and remove callbacks through the `connect()` and `disconnect()` methods. Owners can emit the event through `emit()`.

Parameters

- **eventdoc** – Clients can provide info about the event signature and what it represents. It serves no functional purpose but is useful for readability.
- **onerror** – Callable that takes (etype, evalue, tb) and is fired when any delegate errors.

class `brennivin.threadutils.`**TimerExt**(*interval*, *function*, *args=()*, *kwargs=None*)
Extends the interface of `threading.Timer` to allow for a `restart()` method, which will restart the timer. May be extended in other ways in the future.

Parameters

- **interval** – Number > 0.
- **function** – `function(*args, **kwargs)` that is called when the timer elapses.

restart()
Resets the timer. Will raise if the timer has finished.

class `brennivin.threadutils.`**Token**
Defines a simple object that can be used for callbacks and cancellations.

class brennivin.threadutils.**expiring_memoize**(*expiry=0*, *gettime=None*)

Decorator used to cache method responses evaluated only once within an expiry period (seconds).

Usage:

```
import random
class MyClass(object):
    # Create method whose value is cached for ten minutes
    @ExpiringMemoize(expiry=600)
    def randint(self):
        return random.randint(0, 100)
```

brennivin.threadutils.**join_timeout**(*thread*, *timeout=8*, *errtype=<type 'exceptions.RuntimeError'>*)

`threading.Thread.join(timeout)()` and raises `errtype` if `threading.Thread.is_alive()` after join.

brennivin.threadutils.**memoize**(*func=None*, *uselock=False*, *_lockcls=<IGNORE>*)

Decorator for parameterless functions, to cache their return value. This is functionally the same as using a Lazy instance but allows the use of true functions instead of attributes.

Consider allowing memoization of parameterful functions, but that is far more complex so we don't need to do it right now.

Parameters

- **func** – Filled when used parameter-less, which will use a non-locking memoize (so there is a potential for the function to be called several times). If func is passed, `useLock` *must* be False.
- **uselock** – If True, acquire a lock when evaluating func, so it can only be evaluated once.

1.12 brennivin.traceback2 module

Utility functions for better traceback It has the same interface as the good old `traceback` module.

Appropriate functions have the additional `show_locals` argument which will cause us to try to display local variables for each frame.

Also, the stack extraction functions such as `print_stack()` have an `up` argument used to trim the deepest levels of a callstack, such as when they are called from a utility function in which we aren't interested.

brennivin.traceback2.**extract_stack**(*f=None*, *limit=None*, *up=0*, *extract_locals=0*)

brennivin.traceback2.**extract_tb**(*tb*, *limit=None*, *extract_locals=0*)

brennivin.traceback2.**format_exc**(*limit=None*, *show_locals=0*, *format=0*)

brennivin.traceback2.**format_exception**(*etype*, *value*, *tb*, *limit=None*, *show_locals=0*, *format=0*)

brennivin.traceback2.**format_list**(*extracted_list*, *show_locals=0*, *format=0*)

brennivin.traceback2.**format_stack**(*f=None*, *limit=None*, *up=0*, *show_locals=0*, *format=0*)

brennivin.traceback2.**format_tb**(*tb*, *limit=None*, *show_locals=0*, *format=0*)

brennivin.traceback2.**print_exc**(*limit=None*, *file=None*, *show_locals=0*, *format=0*)

brennivin.traceback2.**print_exception**(*etype*, *value*, *tb*, *limit=None*, *file=None*, *show_locals=0*, *format=0*) `brennivin.traceback2.`**`print_stack`**(*f=None*, *limit=None*, *up=0*, *show_locals=0*, *format=0*, *file=None*)

`brennivin.traceback2.`**`print_tb`**(*tb*, *limit=None*, *file=None*, *show_locals=0*, *format=0*)

1.13 brennivin.yamlext module

This module provides some utility functions for yaml, and automatically chooses the fastest yaml loader/dumper available automatically.

Traditional yaml usage can be replaced as follows:

1.13.1 Dumping to a file

```
with open(path, 'w') as f:
    yaml.dump(obj, f)
=>
yamlext.dumpfile(obj, f)
```

1.13.2 Dumping to a string

```
s = yaml.dump(obj)
=>
s = yamlext.dumps(obj)
```

1.13.3 Dumping to a stream

Almost never a need to use this directly:

```
yaml.dump(obj, stream)
=>
yamlext.dump(obj, stream)
```

1.13.4 Loading from a file

```
with open(path) as f:
    obj = yaml.load(f)
=>
obj = yamlext.loadfile(path)
```

1.13.5 Loading from a string

```
obj = yaml.load(s)
=>
obj = yamlext.loads(s)
```

1.13.6 Loading from a stream

Almost never a need to use this directly:

```
obj = yaml.load(stream)
=>
obj = yamlext.load(stream)
```

Members

brennivin.yamlext.**dumps**(*obj*, ***kwargs*)

brennivin.yamlext.**dumpfile**(*obj*, *path*, ***kwargs*)

brennivin.yamlext.**dump**(*obj*, *stream*, ***kwargs*)

brennivin.yamlext.**loads**(*s*)

brennivin.yamlext.**loadfile**(*path*)

brennivin.yamlext.**load**(*stream*)

class brennivin.yamlext.**PyIO**

> **dump**(*obj*, *stream*, ***kwargs*)
>
> **dumpfile**(*obj*, *path*, ***kwargs*)
>
> **dumps**(*obj*, ***kwargs*)
>
> **load**(*stream*)
>
> **loadfile**(*path*)
>
> **loads**(*s*)

1.14 brennivin.zipfileutils module

Utilities for working with zip files.

1.14.1 Members

class brennivin.zipfileutils.**ZipFile**

> `zipfile.ZipFile` with a context manager (2.6 does not have a context manager. Use if your code needs a ZipFile context manager and must run under 2.6.

brennivin.zipfileutils.**compare_zip_files**(*z1*, *z2*)

> Compares the contents of two zip files (file paths and crcs).
>
> **Returns** None if they are the same.
>
> **Raises FileComparisonError** If the files are different. Message contains a string summarizing the difference.

brennivin.zipfileutils.**is_inside_zipfile**(*filepath*)

> Iterates up a directory tree checking at each level if the path exists. If a subpath exists and points to a zipfile, return true

Parameters filepath – Fully qualified path to a file

`brennivin.zipfileutils.`**`write_dir`**(*rootpath*, *zfile*, *include=ALL*, *exclude=NONE*, *subdir=None*)
Zip all files under `rootpath` to a zip stream. See `zip_dir()` for arguments.

`brennivin.zipfileutils.`**`write_files`**(*fullpaths*, *zfile*, *include=ALL*, *exclude=NONE*, *subdir=None*, *rootpath=None*)
Zip files to a zip stream. See `zip_dir()` for arguments.

Parameters

- **fullpaths** – Absolute paths to files to zip.
- **subdir** – See `zip_dir()`. Only valid if rootpath is also specified.
- **rootpath** – If provided, paths in the archive will be relative to this. Ie, passing in a branch's root and the absolute paths to files in the branch would make the paths in the archive be relative to the branch root.

`brennivin.zipfileutils.`**`zip_dir`**(*rootdir*, *outfile*, *include=ALL*, *exclude=NONE*, *subdir=None*)
Zip all files under the root directory to a zip file at `outfile`.

Parameters

- **outfile** – Path to zipfile, or `ZipFile` stream.
- **include** – Include only files that this function returns True for.
- **exclude** – Include no files that this function returns True for.
- **subdir** – If provided, nest the `rootdir` under this folder in the archive. For example, zipping the directory `spam` with the files `/spam/eggs/ham.txt` and `subdir` of `foo` would yield the archive file `foo/eggs/ham.txt`.

CHAPTER 2

About the name

You can learn about the Icelandic schnapps called “brennivin” on Wikipedia. It literally translates to “burnt wine.”

The reason for this library’s name is that CCP developed an Open Source framework named sake while working on Dust514 (the game was primarily developed in China). As EVE was developed in Iceland (remember brennivin is mostly utilities harvested from EVE), I thought “brennivin” was a fitting name for this library.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*