dtlibs-doc Documentation

Release 0.5.0pre

David Townshend

December 27, 2014

Contents

dtlibs is a Python package designed to support rapid development of desktop applications using PyQt4. It consists of an assortment of modules which can be used together to quickly produce a desktop application with toolbars, menus and undo/redo functionality.

CHAPTER 1

Package Contents

dtlibs.core Provides some general core functions. This module can be loosely compared with the python builtins.

dtlibs.xcollections Provides a few container objects. This is comparable to the python `collections` module

dtlibs.xitertools Iterators extending the capabilities of `itertools`.

dtlibs.xos Path related functions. This extends the capabilities of `os`, `os.path` and `shutil`

dtlibs.dev Some development utilities such as a graphical profiler.

CHAPTER 2

Download and Install

The latest version of **dtlibs** can be downloaded from http://bitbucket.org/aquavitae/dtlibs/downloads and requires Python 3.3 to run.

Table of Contents

3.1 dtlibs.core

This module contains an assortment of useful functions and classes and can be seen as an extension to the python builtin functions.

3.1.1 Functions

`dtlibs.core.`**`true`**(**args*, ***kwargs*)
Always returns `True`, regardless of input. This is useful in functions which require a callable (e.g. `filter`) and is the same as `lambda *args, **kwargs: True`, but reads cleaner.

`dtlibs.core.`**`false`**(**args*, ***kwargs*)
Always returns `False`, regardless of input. This is useful in functions which require a callable (e.g. `filter`) and is the same as `lambda *args, **kwargs: False`, but reads cleaner.

`dtlibs.core.`**`none`**(**args*, ***kwargs*)
Always returns `None`, regardless of input. This is useful in functions which require a callable (e.g. `filter`) and is the same as `lambda *args, **kwargs: None`, but reads cleaner.

`dtlibs.core.`**`isnumeric`**(*value*)
Return true if *value* can be converted to any number type, e.g.:

```
>>> isnumeric('-3e2+4j')
True
>>> isnumeric('string')
False
```

`dtlibs.core.`**`iscallable`**(*value*)
Return True if *value* has a *__call__* member. This includes functions, methods, lambdas, etc.

`dtlibs.core.`**`float2`**(*s*[, *default=0.0*])
Convert any value to a float, or return *default* if the conversion fails.

Parameters

- **s** – Value to convert to a float.
- **default** – This will be returned if *s* could not be converted.

default may be anything, it does not have to be a float. E.g.:

```
>>> print(float2('not a number', None))
None
```

`dtlibs.core.`**`int2`**(*s*[, *default=0*])
Convert any value to an int, or return *default* if the conversion fails.

Parameters

- **s** – Value to convert to an int.
- **default** – This will be returned if *s* could not be converted.

default may be anything, it does not have to be an int. E.g.:

```
>>> print(int2('not a number', None))
None
```

3.1.2 Metaclasses

dtlibs.core.**singleton**(**classes*)

Create a singleton-type metaclass which inherits from all metaclasses of *classes*. Providing arguments to this is only necessary when the singleton class inherits from a non-type base. For example, if `Object` is a special class who's metaclass is not type, then a singleton subclass can be created thus:

class SingletonSubclass(Object, metaclass=singleton(Object)): ...

Subclasses will inherit this metaclass, so use with care at the top of a hierarchy. However, subclasses are handled independently, so

```
>>> class A(metaclass=singleton()):
...     pass
>>> class B(A):
...     pass
>>> A() is B()
False
```

dtlibs.core.**uniqueinstance**(**classes*[, *call_init=False*])

Create a metaclass which only alows unique instances.

Parameters

- **classes** – Classes which the new uniqueinstance metaclass should inherit. They should all be of type `type`.
- **call_init** – Sets whether *__init__* is called for instances which have already been created.

Returns A metaclass object.

The returned metaclass ensures that each instance of the class it creates is unique, based on the initialisation parameters (excluding *self*), which should be hashable.

The actual value used to compare may be set by using callable annotations on the *__init__* method. The callables should take a single argument (the value passed through the parameter) and return a hashable value which will be used as the key.

Variables arguments (e.g. `*args` or `**kwargs`) are handled as a single argument. For example, the key for the argument `*args:type` will always be `tuple`, and the key for the arguments described by `__init__(self, name, *args)` and called as `__init__('spam', 'eggs', 42)` would be ``('spam', ('eggs', 42)) An annotation should always be used with variable keyword arguments since a `dict` is not hashable.

Here is a simple example:

```
>>> class A(metaclass=uniqueinstance()):
...     def __init__(self, id, other_arg:type):
...         pass
>>> A(1, 2) is A(1, 3)
True
>>> A(1, 2) is A('not 1', 2)
False
>>> A(1, 2) is A(1, 'other arg')
False
```

3.1.3 Decorators & Function Tools

`@dtlibs.core.`**`deprecated`**(*version*, *msg*)

A decorator which displays a deprecated warning every call of a callable.

This decorator accepts string arguments as the version and message, and returns the callable with its docstring updated to include a deprecated message. It should not be used on a class since it will result in incorrect name binding, meaning that isinstance checks will fail. Instead, use it on the class' *__init__* method.

For example:

```
>>> @deprecated('1.3', 'use a lumberjack instead')
... def barber(count):
...     "I'm a barber"
...     return count
>>> barber(4)
4
>>> help(barber)
Help of function barber

barber(*args, **kwargs)
    .. deprecated:: 1.3
       use a lumberjack instead

    I'm a barber
```

`@dtlibs.core.`**`info`**(***kwargs*)

This decorator creates an *__info__* property to a function, and stores all keyword arguments passed into it. For example

```
>>> @info(internal_data='Spam', more_data=['spam', 'eggs'])
... def spam_function(spam_count):
...     print(spam_count)
>>> spam_function.__info__
{'internal_data': 'Spam', 'more_data': ['spam', 'eggs']}
```

This has no effect on the function itself, it is merely a way to associate data with a function definition.

`dtlibs.core.`**`hasinfo`**(*obj*[, *name=None*])

Return `True` if *obj* has *name* in its *__info__* dict.

If *name* is None, then `True` if returned if *object* has an *__info__* dict. In either case, if *object* does not have an *__info__* dict, `False` is returned.

Note: This (and the related `getinfo` and `setinfo`) can be used on any objects, not just functions. However, since the info is stored in the *__info__* attribute of the object, use with objects which already make use of this name could cause expected behaviour.

`dtlibs.core.`**`setinfo`**(*obj*[, ***kwargs*])

Set info on a function using `info()`.

This is the similar to `obj = info(**kwargs)(obj)`, except that it updates *__info__* if it already exists.

`dtlibs.core.`**`getinfo`**(*obj*, *name*)

Get the info set on a function (e.g. by `info`).

This is the same as `obj.__info__[name]`. If this does not exist, an exception is raised.

3.2 dtlibs.dev

This module contains some development tools, incuding a graphical profiler and a timer.

3.2.1 Members

class dtlibs.dev.**profile**([*filename=None*])
Profile a function, to observer its performance.

This provides a decorator interface to cProfile. It does not add any functionality, but rather makes it easier to quickly profile an existing function during testing without needing a large amount of boilerplate.

The simplest way of using this is:

```
@profile()
def long_spam_song(spam):
    for s in spam:
        sing_spam_song(s)
```

This is similar to calling:

```
def long_spam_song(spam):
    def run(spam):
        for s in spam:
            sing_spam_song(s)
    cProfile.run('run()', sort='calls')
```

A more complex usage is:

```
prof = profile('long_spam_song.profile')
prof.sort_stats('time').print_callers(10)
@prof
def long_spam_song(spam):
    for s in spam:
        sing_spam_song(s)
```

The argument to profile is the name of the file to write profile information to (see the cProfile documentation for more detail). The additional methods called on this are the same as those provided by pstats.Stats. If no additional methods are used, .sort_stats('calls').print_stats(10) is applied. Note that if any method is used, one of the *print* methods must also be used to display the results.

profile can also be used as a context manager:

with profile('spam_song'): sing_spam_song()

class dtlibs.dev.**Theme**
A Colour theme to use in graphs.

Parameters

- **bgcolor** – Global background colour (default is white)
- **mincolor** – Minimum node colour (default is black)
- **maxcolor** – Maximum node colour (default is white)
- **fontname** – Fontname for text (default is 'Arial')
- **minfontsize** – Minimum font size (default is 10)
- **maxfontsize** – Maximum font size (default is 10)
- **minpenwidth** – Minimum pen width for links (default is 0.5)
- **maxpenwidth** – Maximum pen with for links (default is 4.0)

- **gamma** – Gamma correction (default is 2.2)
- **skew** – Skew the colour curve (Default is 1.0)

Colours are specified as RGB tuples.

The following themes are predefined.

`dtlibs.dev.`**`temp_theme`**
A colour theme ranging from red to blue.

`dtlibs.dev.`**`pink_theme`**
A pink colour theme.

`dtlibs.dev.`**`gray_theme`**
A gray colour theme

`dtlibs.dev.`**`mono_theme`**
A black and white colour theme

`graph(filename, [fmt='pdf', nodethres=0.5, edgethres=0.1, theme='temp_theme', strip=False, wrap=False])`
This is similar to `profile`, but shows a graph instead.

Parameters

- **filename** – The name of the file (including extension) to write.
- **fmt** – The file format as used by dot's '-T' parameter
- **nodethres** – Eliminate nodes below this threshold fraction
- **edgethres** – Eliminate edges below this threshold fraction
- **theme** – Colour theme (see `Theme`)
- **strip** – Strip mangling from C++ function names
- **wrap** – Wrap function names

This requires <graphviz http://www.graphviz.org/>_ and <gprof2dot http://code.google.com/p/jrfonseca/wiki/Gprof2Dot>_.

3.3 dtlibs.xcollections

`dtlibs.xcollections.`**`nameddict`**(*name*, **keys*, ***types*)
Factory for DefinedDict.

A new DefinedDict subclass is created with keys. The key types may optionally be specified using types. An example is:

```
>>> Dict = nameddict('Dict', 'key1 key2 key3', key1=str, key2=list)
>>> d = Dict()
>>> d.key1 = 'a string'
>>> d.key2 = ['a', 'list']
>>> d.key3 = 'anything'
```

`dtlibs.xcollections.`**`typedlist`**(*name*, *type_*)
Factory function for specific TypedList object.

While TypeLists can be created directly, this function constructs a new TypeList subclass for a specific type, allowing it to be reused.

```
>>> StringList = typedlist('StringList', str)
>>> names = StringList()
>>> addresses = StringList()
>>> names.append('Mike')
```

```
>>> names.append(34)
Traceback (most recent call last):
    ...
TypeError: 34 is not type <class 'str'>
Traceback (most recent call last):
    ...
TypeError: 34 is not type <class 'str'>
```

class dtlibs.xcollections.**TypedList**(*type_*, *initial=None*)
A list that contains a specific type of object.

appendnew(**args*, ***kwargs*)
Create a new instance the list type and append it.

Since the type of data stored by the list is known, a new object can be added to it by simply specifying the constructor arguments. The return value is the newly created object.

For example:

```
>>> import datetime
>>> l = TypedList(datetime.date)
>>> obj = l.appendnew(2001, 1, 1)
>>> print(obj)
2001-01-01
```

is the same as

```
>>> import datetime
>>> l = TypedList(datetime.date)
>>> obj = datetime.date(2001, 1, 1)
>>> l.append(obj)
```

insert(*index*, *value*)

type()

class dtlibs.xcollections.**SelectList**(*iterable=None*)
A list of which some items are selected.

The example below illustrates basic usage of this class.

```
>>> l = SelectList([1, 2, 3, 4, 5])
>>> l.select(3)
[1, 2, <3>, 4, 5]
>>> l.selection
(3,)
>>> l.select(6)
Traceback (most recent call last):
    ...
ValueError: 6
>>> l
[1, 2, <3>, 4, 5]
>>> l.clear()
>>> l.selection
()
>>> l.indexselect(0, 2)
[<1>, 2, <3>, 4, 5]
>>> l.indexselection
(0, 2)
>>> l.select(5, 3)
[<1>, 2, <3>, 4, <5>]
>>> l.selection
(1, 3, 5)
Traceback (most recent call last):
    ...
ValueError: 6
```

The following rules govern the processing of the selections:

1. If options contains duplicates, the first matching value is selected when using `select`

```
>>> l = SelectList([1, 2, 1])
>>> l.select(1)
[<1>, 2, 1]
```

However, it is possible to explicitly select a value using indexselect().

```
>>> l = SelectList([1, 2, 1])
>>> l.indexselect(2)
[1, 2, <1>]
```

2. The selection lists are unordered.

```
>>> l = SelectList([1, 2, 3, 4, 5])
>>> _ = l.select(3)
>>> _ = l.select(1)
>>> l.selection
(1, 3)
```

3. If a selected value is changed in the SelectList, it is removed from the selection.

```
>>> l = SelectList([1, 2, 3])
>>> l.select(2)
[1, <2>, 3]
>>> l[1] = 4
>>> l
[1, 4, 3]
```

clear()
Clear the selection.

indexselect(**indexes*)
Select by indexes and return the SelectList.

indexselection
Return a tuple of selected indexes.

indexunselect(**indexes*)
Remove values by index and return the SelectList.

```
>>> l = SelectList('penguin')
>>> l.select('p', 'g')
[<'p'>, 'e', 'n', <'g'>, 'u', 'i', 'n']
>>> l.indexunselect(3)
[<'p'>, 'e', 'n', 'g', 'u', 'i', 'n']
>>> l.indexunselect(4)
Traceback (most recent call last):
    ...
IndexError: Item at position '4' is not selected
Traceback (most recent call last):
    ...
IndexError: Item at position '4' is not selected
```

insert(*index*, *value*)

select(**values*)
Select a group of values and return the SelectList.

```
>>> l = SelectList('abcdefg')
>>> l.select('a', 'd', 'f')
[<'a'>, 'b', 'c', <'d'>, 'e', <'f'>, 'g']
```

selection
Return a tuple of the selected values.

unselect (**values*)
Remove values from the selection list and return the SelectList.

If the selection contains duplicate values, then the first found is removed. If no matching values are found, then an exception is raised.

```
>>> l = SelectList('penguin')
>>> l.indexselect(2, 6)
['p', 'e', <'n'>, 'g', 'u', 'i', <'n'>]
>>> l.unselect('n')
['p', 'e', 'n', 'g', 'u', 'i', <'n'>]
>>> l.unselect('p')
Traceback (most recent call last):
    ...
ValueError: 'p' is not selected
Traceback (most recent call last):
    ...
ValueError: 'p' is not selected
```

class dtlibs.xcollections.**FilterList** (*key*, *parent*)
A filtered wrapper around a list.

This class wraps around a list, but filters all output. This is similar to using the built-in `filter` function, but allows changes through it to the underlying list. This is not as fast as filter, though, so should only be used when modifications are needed.

Initialisation follows the same format as `filter`:

```
>>> key = lambda n: n in [1, 3, 5, 7, 9]
>>> parent = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> f = FilterList(key, parent)
```

Iterating over the list returns only those values for which *key* returns True.

```
>>> [i for i in f]
[1, 3, 5, 7, 9]
```

Similarly, item indexes work according to this.

```
>>> f[3]
7
```

Assignments, insertions and deletions are supported, and affect the *parent*.

```
>>> f[3] = 1
>>> f.insert(4, 3)
>>> del f[1]
>>> parent
[1, 2, 4, 5, 6, 1, 8, 3, 9, 10]
```

Note that slices can only be used when querying. They cannot be used for assignments of deletions.

insert (*index*, *value*)

class dtlibs.xcollections.**StrictList** (*accepts*, *initial=None*)
A list that is fussy about what it contains.

A `StrictList` may only contain object for which `accepts(obj)` is True.

```
>>> largenumbers = StrictList(lambda n: n > 1000, range(2000, 2005))
>>> print(list(largenumbers))
[2000, 2001, 2002, 2003, 2004]
>>> largenumbers.append(3)
Traceback (most recent call last):
  ...
ValueError: 3
Traceback (most recent call last):
```

```
  ...
ValueError: 3
```

accepts(*value*)
Return False if *value* is not accepted.

insert(*index*, *value*)

class dtlibs.xcollections.**MultiDict**(**args*, ***kwargs*)
A dictionary which allows multiple keys.

get(*key*, *default=<class 'dtlibs.xcollections._multidict._Unused'>*)

items(*key=<class 'dtlibs.xcollections._multidict._Unused'>*)

keys()

popitem(*pair=None*)

values(*key=<class 'dtlibs.xcollections._multidict._Unused'>*)

3.4 dtlibs.xitertools

dtlibs.xitertools.**compact**(*it*)
Iterate through a sequence, skipping duplicates.

```
>>> seq = 'MINIMUM'
>>> print([i for i in compact(seq)])
['M', 'I', 'N', 'U']
```

dtlibs.xitertools.**group**(*iterable*, *size*, *test*)
Group items in iterable in tuples of *size* if *test* is `True`.

Each consecutive slice of *size* items of *iterable* are tested by calling *test(items)*. It *test* returns true, then they are grouped in a tuple.

```
>>> def ends_with_list(items):
...     return isinstance(items[1], list)
>>> [i for i in group([1, [2, 3], 4, 5, [6, 7, 8]], 2, ends_with_list)]
[(1, [2, 3]), 4, (5, [6, 7, 8])]
```

3.5 dtlibs.xos

A collection of functions which extend `os`, `os.path` and `shutil`.

class dtlibs.xos.**Filename**(*name*, *relative=None*)
Intelligently handle a file path.

The filename is broken down into `basepath/relpath/basename.extension`. *basepath* and *relpath* always end in a trailing slash, and *extension*, if it exists starts with a `.`. This means that the the full path is always `basepath + relpath + basename + extension`.

If *relpath* or *extension* are not used, then it they are an empty string.

In addition, the following properties are supported:

Property	Return value
fullname	basepath/relpath/basename.extension
relname	relpath/basename.extension
path	basepath/relpath
filename	basename.extension

The extension is interpreted to be at the first `.`. All paths returned are normalised using `normpath()`.

Examples:

```
>>> path = Filename('/tmp/folder/and/file.ext', '/tmp/folder')
>>> path.basepath
'/tmp/folder/'
>>> path.relname
'and/file.ext'
>>> path.fullname
'/tmp/folder/and/file.ext'
>>> path.path
'/tmp/folder/and/'
>>> path.filename
'file.ext'
>>> Filename('/path/file').extension
''
>>> Filename('/path/file').fullname
'/path/file'
```

basename

basepath

exists()

extension

filename

fullname

path

relname

relpath

dtlibs.xos.**create**(*file*, *mode=None*)
Similar to builtin `open()`, but raise an exception if file exists.

The mode can be specified as for `open()`, but cannot contain read, write, update or append modes, since the file created is always opened in write mode.

dtlibs.xos.**getfiles**(*path*)
Return a generator of `Filename` objects for each file in *path*.

Each Filename object returned represents a single file in a tree relative to *path*. Directories are ignored.

dtlibs.xos.**hasroot**(*path*, *root*)
Return true if *path* has the root specified.

```
>>> hasroot('/usr/share/doc/python32', '/usr/share')
True
>>> hasroot('../folder', '../')
True
>>> hasroot('/usr/share', '/usr/sh')
False
```

dtlibs.xos.**iswindowspath**(*path*)
Return true if path is recognised as having a drive letter.

```
>>> iswindowspath('c:\windows')
True
>>> iswindowspath('c:/windows')
True
>>> iswindowspath('/usr/share')
False
```

dtlibs.xos.**levels**(*path*, *root=None*)
Return the number of levels in the path relative to *root*.

The following conditions are fulfilled:

1. If root is missing, then path is not relative to anything (even if it is a relative path), and levels is always a number with the exception of cases defined by 5. below.
2. If root is absolute and path is relative, then paths is assumed to be relative to root. In this case `levels(path, root)` gives the same result as `levels(path, None)`.
3. If root is relative then path must also be relative, or None is returned.
4. If root and path are both relative then they are assume to be relative to the same position. i.e. `levels('share/doc', 'lib')` is the same as `levels('/share/doc', '/lib')`.
5. If the final path, relative to root, has negative levels (e.g. '../share'), then *None* will be returned.
6. On windows, the drive letter is used in the comparison, but is not counted as a level.

dtlibs.xos.**normpath**(*path*)

Similar to `os.path.normpath()`, but not dependant on OS.

This always returns the same string, regardless of operating system. Note that this can result in an incorrect path on unusual systems.

For example, the following should work on any system:

```
>>> normpath(r'\usr/local')
'/usr/local'
>>> normpath(r'c:\windows\system')
'c:/windows/system'
```

dtlibs.xos.**remove_empty**(*root*)

Remove all empty directories in *root*, including subfolders.

class dtlibs.xos.**safewriter**(*name*, *text=False*, *backup=None*)

Provide a safe environment for writing files.

`safewriter` is a context manager which acts similar to the file object returned by `open()`. However, it works by writing to a temporary file first, so that data is not lost if for any reason the write fails.

`safewriter` is typically instantiated as a context manager:

```
with safewriter('file') as f:
    f.write(b'spam')
```

If an exception occurs before the context manager exits, the temporary file is removed and the original data left untouched. If the write is a success, the temporary file is renamed to the target filename, after optionally backing the target up.

This is all done as safely as possible, given the provisions of `tempfile.mkstemp()` and `os.rename()`. Under normal circumstances, the only possibility of a race condition is that a new file with the same name as the target could be created after the target is removed and before the temporary file is renamed. This will only be possible on certain platforms where `os.rename()` does not automatically overwrite.

abort()

close()

name

Return the name of the file to be written to.

This allows for writing to files using custom methods requiring a name, e.g. as a sqlite database:

```
with safewriter('database.sqlite') as f:
    conn = sqlite.connect(f.name)
    ... # write stuff to database
    conn.close()
```

dtlibs.xos.**walkstats**(*root*)

Iterate over the dirs in path and yield *(base, stats)* tuples.

Each stats object is itself a dict of *{path: stats}* within *root*. *path* is the relative path, but is not recursive. i.e. the first iteration of `walkstats('/usr')` would include, for example, '/usr/share' but not '/usr/share/doc'

CHAPTER 4

Indices and tables

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

d

R

S

T

U

V

W