functools — Tools for Manipulating Functions

Purpose: Functions that operate on other functions.

The `functools` module provides tools for adapting or extending functions and other callable objects, without completely rewriting them.

Decorators

The primary tool supplied by the `functools` module is the class `partial`, which can be used to “wrap” a callable object with default arguments. The resulting object is itself callable and can be treated as though it is the original function. It takes all of the same arguments as the original, and can be invoked with extra positional or named arguments as well. A `partial` can be used instead of a `lambda` to provide default arguments to a function, while leaving some arguments unspecified.

Partial Objects

This example shows two simple `partial` objects for the function `myfunc()`. The output of `show_details()` includes the `func`, `args`, and `keywords` attributes of the partial object.

```
# functools_partial.py

import functools


def myfunc(a, b=2):
    "Docstring for myfunc()."
    print('  called myfunc with:', (a, b))


def show_details(name, f, is_partial=False):
    "Show details of a callable object."
    print('{}:'.format(name))
    print('  object:', f)
    if not is_partial:
        print('  __name__:', f.__name__)
    if is_partial:
        print('  func:', f.func)
        print('  args:', f.args)
        print('  keywords:', f.keywords)
    return


show_details('myfunc', myfunc)
myfunc('a', 3)
print()

# Set a different default value for 'b', but require
# the caller to provide 'a'.
p1 = functools.partial(myfunc, b=4)
show_details('partial with named default', p1, True)
p1('passing a')
p1('override b', b=5)
print()

# Set default values for both 'a' and 'b'.
p2 = functools.partial(myfunc, 'default a', b=99)
show_details('partial with defaults', p2, True)
p2()
p2(b='override b')
print()

print('Insufficient arguments:')
p1()
```

At the end of the example, the first partial created is invoked without passing a value for a, causing an exception.

```
$ python3 functools_partial.py

myfunc:
  object: <function myfunc at 0x1007a6a60>
  __name__: myfunc
  called myfunc with: ('a', 3)

partial with named default:
  object: functools.partial(<function myfunc at 0x1007a6a60>,
b=4)
  func: <function myfunc at 0x1007a6a60>
  args: ()
  keywords: {'b': 4}
  called myfunc with: ('passing a', 4)
  called myfunc with: ('override b', 5)

partial with defaults:
  object: functools.partial(<function myfunc at 0x1007a6a60>,
'default a', b=99)
  func: <function myfunc at 0x1007a6a60>
  args: ('default a',)
  keywords: {'b': 99}
  called myfunc with: ('default a', 99)
  called myfunc with: ('default a', 'override b')

Insufficient arguments:
Traceback (most recent call last):
  File "functools_partial.py", line 51, in <module>
    p1()
TypeError: myfunc() missing 1 required positional argument: 'a'
```

Acquiring Function Properties

The partial object does not have __name__ or __doc__ attributes by default, and without those attributes, decorated functions are more difficult to debug. Using update_wrapper(), copies or adds attributes from the original function to the partial object.

```
# functools_update_wrapper.py

import functools


def myfunc(a, b=2):
    "Docstring for myfunc()."
    print('  called myfunc with:', (a, b))


def show_details(name, f):
    "Show details of a callable object."
    print('{}:'.format(name))
    print('  object:', f)
    print('  __name__:', end=' ')
    try:
        print(f.__name__)
    except AttributeError:
        print('(no __name__)')
    print('  __doc__', repr(f.__doc__))
    print()


show_details('myfunc', myfunc)

p1 = functools.partial(myfunc, b=4)
show_details('raw wrapper', p1)

print('Updating wrapper:')
print('  assign:', functools.WRAPPER_ASSIGNMENTS)
print('  update:', functools.WRAPPER_UPDATES)
```

```
print('  update:', functools.WRAPPER_UPDATES)
print()

functools.update_wrapper(p1, myfunc)
show_details('updated wrapper', p1)
```

The attributes added to the wrapper are defined in WRAPPER_ASSIGNMENTS, while WRAPPER_UPDATES lists values to be modified.

```
$ python3 functools_update_wrapper.py

myfunc:
  object: <function myfunc at 0x1018a6a60>
  __name__: myfunc
  __doc__ 'Docstring for myfunc().'

raw wrapper:
  object: functools.partial(<function myfunc at 0x1018a6a60>,
b=4)
  __name__: (no __name__)
  __doc__ 'partial(func, *args, **keywords) - new function with
partial application\n    of the given arguments and keywords.\n'

Updating wrapper:
  assign: ('__module__', '__name__', '__qualname__', '__doc__',
'__annotations__')
  update: ('__dict__',)

updated wrapper:
  object: functools.partial(<function myfunc at 0x1018a6a60>,
b=4)
  __name__: myfunc
  __doc__ 'Docstring for myfunc().'
```

Other Callables

Partials work with any callable object, not just with standalone functions.

```
# functools_callable.py

import functools


class MyClass:
    "Demonstration class for functools"

    def __call__(self, e, f=6):
        "Docstring for MyClass.__call__"
        print('  called object with:', (self, e, f))


def show_details(name, f):
    "Show details of a callable object."
    print('{}:'.format(name))
    print('  object:', f)
    print('  __name__:', end=' ')
    try:
        print(f.__name__)
    except AttributeError:
        print('(no __name__)')
    print('  __doc__', repr(f.__doc__))
    return


o = MyClass()

show_details('instance', o)
o('e goes here')
print()

p = functools.partial(o, e='default for e', f=8)
functools.update_wrapper(p, o)
```

```
functools.update_wrapper(p, o)
show_details('instance wrapper', p)
p()
```

This example creates partials from an instance of a class with a `__call__()` method.

```
$ python3 functools_callable.py

instance:
  object: <__main__.MyClass object at 0x1011b1cf8>
  __name__: (no __name__)
  __doc__ 'Demonstration class for functools'
  called object with: (<__main__.MyClass object at 0x1011b1cf8>,
'e goes here', 6)

instance wrapper:
  object: functools.partial(<__main__.MyClass object at
0x1011b1cf8>, f=8, e='default for e')
  __name__: (no __name__)
  __doc__ 'Demonstration class for functools'
  called object with: (<__main__.MyClass object at 0x1011b1cf8>,
'default for e', 8)
```

Methods and Functions

While `partial()` returns a callable ready to be used directly, `partialmethod()` returns a callable ready to be used as an unbound method of an object. In the following example, the same standalone function is added as an attribute of `MyClass` twice, once using `partialmethod()` as `method1()` and again using `partial()` as `method2()`.

```
# functools_partialmethod.py

import functools


def standalone(self, a=1, b=2):
    "Standalone function"
    print('  called standalone with:', (self, a, b))
    if self is not None:
        print('  self.attr =', self.attr)


class MyClass:
    "Demonstration class for functools"

    def __init__(self):
        self.attr = 'instance attribute'

    method1 = functools.partialmethod(standalone)
    method2 = functools.partial(standalone)


o = MyClass()

print('standalone')
standalone(None)
print()

print('method1 as partialmethod')
o.method1()
print()

print('method2 as partial')
try:
    o.method2()
except TypeError as err:
    print('ERROR: {}'.format(err))
```

`method1()` can be called from an instance of `MyClass`, and the instance is passed as the first argument just as with methods defined normally. `method2()` is not set up as a bound method, and so the `self` argument must be passed explicitly, or the call will result in a `TypeError`.

```
$ python3 functools_partialmethod.py

standalone
  called standalone with: (None, 1, 2)

method1 as partialmethod
  called standalone with: (<__main__.MyClass object at
0x1007b1d30>, 1, 2)
  self.attr = instance attribute

method2 as partial
ERROR: standalone() missing 1 required positional argument:
'self'
```

Acquiring Function Properties for Decorators

Updating the properties of a wrapped callable is especially useful when used in a decorator, since the transformed function ends up with properties of the original “bare” function.

```
# functools_wraps.py

import functools


def show_details(name, f):
    "Show details of a callable object."
    print('{}:'.format(name))
    print('  object:', f)
    print('  __name__:', end=' ')
    try:
        print(f.__name__)
    except AttributeError:
        print('(no __name__)')
    print('  __doc__', repr(f.__doc__))
    print()


def simple_decorator(f):
    @functools.wraps(f)
    def decorated(a='decorated defaults', b=1):
        print('  decorated:', (a, b))
        print('  ', end=' ')
        return f(a, b=b)
    return decorated


def myfunc(a, b=2):
    "myfunc() is not complicated"
    print('  myfunc:', (a, b))
    return


# The raw function
show_details('myfunc', myfunc)
myfunc('unwrapped, default b')
myfunc('unwrapped, passing b', 3)
print()

# Wrap explicitly
wrapped_myfunc = simple_decorator(myfunc)
show_details('wrapped_myfunc', wrapped_myfunc)
wrapped_myfunc()
wrapped_myfunc('args to wrapped', 4)
print()


# Wrap with decorator syntax
@simple_decorator
def decorated_myfunc(a, b):
```

```
    myfunc(a, b)
    return


show_details('decorated_myfunc', decorated_myfunc)
decorated_myfunc()
decorated_myfunc('args to decorated', 4)
```

functools provides a decorator, wraps(), that applies update_wrapper() to the decorated function.

```
$ python3 functools_wraps.py

myfunc:
  object: <function myfunc at 0x101241b70>
  __name__: myfunc
  __doc__ 'myfunc() is not complicated'

  myfunc: ('unwrapped, default b', 2)
  myfunc: ('unwrapped, passing b', 3)

wrapped_myfunc:
  object: <function myfunc at 0x1012e62f0>
  __name__: myfunc
  __doc__ 'myfunc() is not complicated'

  decorated: ('decorated defaults', 1)
     myfunc: ('decorated defaults', 1)
  decorated: ('args to wrapped', 4)
     myfunc: ('args to wrapped', 4)

decorated_myfunc:
  object: <function decorated_myfunc at 0x1012e6400>
  __name__: decorated_myfunc
  __doc__ None

  decorated: ('decorated defaults', 1)
     myfunc: ('decorated defaults', 1)
  decorated: ('args to decorated', 4)
     myfunc: ('args to decorated', 4)
```

Comparison

Under Python 2, classes could define a __cmp__() method that returns -1, 0, or 1 based on whether the object is less than, equal to, or greater than the item being compared. Python 2.1 introduced the *rich comparison* methods API (__lt__(), __le__(), __eq__(), __ne__(), __gt__(), and __ge__()), which perform a single comparison operation and return a boolean value. Python 3 deprecated __cmp__() in favor of these new methods and functools provides tools to make it easier to write classes that comply with the new comparison requirements in Python 3.

Rich Comparison

The rich comparison API is designed to allow classes with complex comparisons to implement each test in the most efficient way possible. However, for classes where comparison is relatively simple, there is no point in manually creating each of the rich comparison methods. The total_ordering() class decorator takes a class that provides some of the methods, and adds the rest of them.

```
# functools_total_ordering.py

import functools
import inspect
from pprint import pprint


@functools.total_ordering
class MyObject:

    def __init__(self, val):
        self.val = val

    def __eq__(self, other):
        print('  testing __eq__({}, {})'.format(
```

```
        print('  testing __eq__({}, {})'.format(
            self.val, other.val))
        return self.val == other.val

    def __gt__(self, other):
        print('  testing __gt__({}, {})'.format(
            self.val, other.val))
        return self.val > other.val


print('Methods:\n')
pprint(inspect.getmembers(MyObject, inspect.isfunction))

a = MyObject(1)
b = MyObject(2)

print('\nComparisons:')
for expr in ['a < b', 'a <= b', 'a == b', 'a >= b', 'a > b']:
    print('\n{:<6}:'.format(expr))
    result = eval(expr)
    print('  result of {}: {}'.format(expr, result))
```

The class must provide implementation of `__eq__()` and one other rich comparison method. The decorator adds implementations of the rest of the methods that work by using the comparisons provided. If a comparison cannot be made, the method should return `NotImplemented` so the comparison can be tried using the reverse comparison operators on the other object, before failing entirely.

```
$ python3 functools_total_ordering.py

Methods:

[('__eq__', <function MyObject.__eq__ at 0x10139a488>),
 ('__ge__', <function _ge_from_gt at 0x1012e2510>),
 ('__gt__', <function MyObject.__gt__ at 0x10139a510>),
 ('__init__', <function MyObject.__init__ at 0x10139a400>),
 ('__le__', <function _le_from_gt at 0x1012e2598>),
 ('__lt__', <function _lt_from_gt at 0x1012e2488>)]

Comparisons:

a < b :
  testing __gt__(1, 2)
  testing __eq__(1, 2)
  result of a < b: True

a <= b:
  testing __gt__(1, 2)
  result of a <= b: True

a == b:
  testing __eq__(1, 2)
  result of a == b: False

a >= b:
  testing __gt__(1, 2)
  testing __eq__(1, 2)
  result of a >= b: False

a > b :
  testing __gt__(1, 2)
  result of a > b: False
```

Collation Order

Since old-style comparison functions are deprecated in Python 3, the `cmp` argument to functions like `sort()` are also no longer supported. Older programs that use comparison functions can use `cmp_to_key()` to convert them to a function that returns a *collation key*, which is used to determine the position in the final sequence.

```
# functools_cmp_to_key.py

import functools
```

```
import functools


class MyObject:

    def __init__(self, val):
        self.val = val

    def __str__(self):
        return 'MyObject({})'.format(self.val)


def compare_obj(a, b):
    """Old-style comparison function.
    """
    print('comparing {} and {}'.format(a, b))
    if a.val < b.val:
        return -1
    elif a.val > b.val:
        return 1
    return 0


# Make a key function using cmp_to_key()
get_key = functools.cmp_to_key(compare_obj)

def get_key_wrapper(o):
    "Wrapper function for get_key to allow for print statements."
    new_key = get_key(o)
    print('key_wrapper({}) -> {!r}'.format(o, new_key))
    return new_key


objs = [MyObject(x) for x in range(5, 0, -1)]

for o in sorted(objs, key=get_key_wrapper):
    print(o)
```

Normally `cmp_to_key()` would be used directly, but in this example an extra wrapper function is introduced to print out more information as the key function is being called.

The output shows that `sorted()` starts by calling `get_key_wrapper()` for each item in the sequence to produce a key. The keys returned by `cmp_to_key()` are instances of a class defined in `functools` that implements the rich comparison API using the old-style comparison function passed in. After all of the keys are created, the sequence is sorted by comparing the keys.

```
$ python3 functools_cmp_to_key.py

key_wrapper(MyObject(5)) -> <functools.KeyWrapper object at
0x1011c5530>
key_wrapper(MyObject(4)) -> <functools.KeyWrapper object at
0x1011c5510>
key_wrapper(MyObject(3)) -> <functools.KeyWrapper object at
0x1011c54f0>
key_wrapper(MyObject(2)) -> <functools.KeyWrapper object at
0x1011c5390>
key_wrapper(MyObject(1)) -> <functools.KeyWrapper object at
0x1011c5710>
comparing MyObject(4) and MyObject(5)
comparing MyObject(3) and MyObject(4)
comparing MyObject(2) and MyObject(3)
comparing MyObject(1) and MyObject(2)
MyObject(1)
MyObject(2)
MyObject(3)
MyObject(4)
MyObject(5)
```

Caching

The `lru_cache()` decorator wraps a function in a least-recently-used cache. Arguments to the function are used to build a hash key, which is then mapped to the result. Subsequent calls with the same arguments will fetch the value from the cache instead of calling the function. The decorator also adds methods to the function to examine the state of the cache (cache_info()) and empty the cache (cache_clear()).

```
# functools_lru_cache.py

import functools


@functools.lru_cache()
def expensive(a, b):
    print('expensive({}, {})'.format(a, b))
    return a * b


MAX = 2

print('First set of calls:')
for i in range(MAX):
    for j in range(MAX):
        expensive(i, j)
print(expensive.cache_info())

print('\nSecond set of calls:')
for i in range(MAX + 1):
    for j in range(MAX + 1):
        expensive(i, j)
print(expensive.cache_info())

print('\nClearing cache:')
expensive.cache_clear()
print(expensive.cache_info())

print('\nThird set of calls:')
for i in range(MAX):
    for j in range(MAX):
        expensive(i, j)
print(expensive.cache_info())
```

This example makes several calls to expensive() in a set of nested loops. The second time those calls are made with the same values the results appear in the cache. When the cache is cleared and the loops are run again the values must be recomputed.

```
$ python3 functools_lru_cache.py

First set of calls:
expensive(0, 0)
expensive(0, 1)
expensive(1, 0)
expensive(1, 1)
CacheInfo(hits=0, misses=4, maxsize=128, currsize=4)

Second set of calls:
expensive(0, 2)
expensive(1, 2)
expensive(2, 0)
expensive(2, 1)
expensive(2, 2)
CacheInfo(hits=4, misses=9, maxsize=128, currsize=9)

Clearing cache:
CacheInfo(hits=0, misses=0, maxsize=128, currsize=0)

Third set of calls:
expensive(0, 0)
expensive(0, 1)
expensive(1, 0)
expensive(1, 1)
CacheInfo(hits=0, misses=4, maxsize=128, currsize=4)
```

To prevent the cache from growing without bounds in a long-running process, it is given a maximum size. The default is 128 entries, but that can be changed for each cache using the maxsize argument.

```
# functools_lru_cache_expire.py

import functools


@functools.lru_cache(maxsize=2)
def expensive(a, b):
    print('called expensive({}, {})'.format(a, b))
    return a * b


def make_call(a, b):
    print('({}, {})'.format(a, b), end=' ')
    pre_hits = expensive.cache_info().hits
    expensive(a, b)
    post_hits = expensive.cache_info().hits
    if post_hits > pre_hits:
        print('cache hit')


print('Establish the cache')
make_call(1, 2)
make_call(2, 3)

print('\nUse cached items')
make_call(1, 2)
make_call(2, 3)

print('\nCompute a new value, triggering cache expiration')
make_call(3, 4)

print('\nCache still contains one old item')
make_call(2, 3)

print('\nOldest item needs to be recomputed')
make_call(1, 2)
```

In this example the cache size is set to 2 entries. When the third set of unique arguments (3, 4) is used the oldest item in the cache is dropped and replaced with the new result.

```
$ python3 functools_lru_cache_expire.py

Establish the cache
(1, 2) called expensive(1, 2)
(2, 3) called expensive(2, 3)

Use cached items
(1, 2) cache hit
(2, 3) cache hit

Compute a new value, triggering cache expiration
(3, 4) called expensive(3, 4)

Cache still contains one old item
(2, 3) cache hit

Oldest item needs to be recomputed
(1, 2) called expensive(1, 2)
```

The keys for the cache managed by lru_cache() must be hashable, so all of the arguments to the function wrapped with the cache lookup must be hashable.

```
# functools_lru_cache_arguments.py

import functools


@functools.lru_cache(maxsize=2)
```

```
def expensive(a, b):
    print('called expensive({}, {})'.format(a, b))
    return a * b


def make_call(a, b):
    print('({}, {})'.format(a, b), end=' ')
    pre_hits = expensive.cache_info().hits
    expensive(a, b)
    post_hits = expensive.cache_info().hits
    if post_hits > pre_hits:
        print('cache hit')


make_call(1, 2)

try:
    make_call([1], 2)
except TypeError as err:
    print('ERROR: {}'.format(err))

try:
    make_call(1, {'2': 'two'})
except TypeError as err:
    print('ERROR: {}'.format(err))
```

If any object that can't be hashed is passed in to the function, a `TypeError` is raised.

```
$ python3 functools_lru_cache_arguments.py

(1, 2) called expensive(1, 2)
([1], 2) ERROR: unhashable type: 'list'
(1, {'2': 'two'}) ERROR: unhashable type: 'dict'
```

Reducing a Data Set

The `reduce()` function takes a callable and a sequence of data as input and produces a single value as output based on invoking the callable with the values from the sequence and accumulating the resulting output.

```
# functools_reduce.py

import functools


def do_reduce(a, b):
    print('do_reduce({}, {})'.format(a, b))
    return a + b


data = range(1, 5)
print(data)
result = functools.reduce(do_reduce, data)
print('result: {}'.format(result))
```

This example adds up the numbers in the input sequence.

```
$ python3 functools_reduce.py

range(1, 5)
do_reduce(1, 2)
do_reduce(3, 3)
do_reduce(6, 4)
result: 10
```

The optional `initializer` argument is placed at the front of the sequence and processed along with the other items. This can be used to update a previously computed value with new inputs.

```
# functools_reduce_initializer.py
```

```
import functools


def do_reduce(a, b):
    print('do_reduce({}, {})'.format(a, b))
    return a + b


data = range(1, 5)
print(data)
result = functools.reduce(do_reduce, data, 99)
print('result: {}'.format(result))
```

In this example a previous sum of 99 is used to initialize the value computed by `reduce()`.

```
$ python3 functools_reduce_initializer.py

range(1, 5)
do_reduce(99, 1)
do_reduce(100, 2)
do_reduce(102, 3)
do_reduce(105, 4)
result: 109
```

Sequences with a single item automatically reduce to that value when no initializer is present. Empty lists generate an error, unless an initializer is provided.

```
# functools_reduce_short_sequences.py
```

```
import functools


def do_reduce(a, b):
    print('do_reduce({}, {})'.format(a, b))
    return a + b


print('Single item in sequence:',
      functools.reduce(do_reduce, [1]))

print('Single item in sequence with initializer:',
      functools.reduce(do_reduce, [1], 99))

print('Empty sequence with initializer:',
      functools.reduce(do_reduce, [], 99))

try:
    print('Empty sequence:', functools.reduce(do_reduce, []))
except TypeError as err:
    print('ERROR: {}'.format(err))
```

Because the initializer argument serves as a default, but is also combined with the new values if the input sequence is not empty, it is important to consider carefully whether to use it. When it does not make sense to combine the default with new values, it is better to catch the `TypeError` rather than passing an initializer.

```
$ python3 functools_reduce_short_sequences.py

Single item in sequence: 1
do_reduce(99, 1)
Single item in sequence with initializer: 100
Empty sequence with initializer: 99
ERROR: reduce() of empty sequence with no initial value
```

Generic Functions

In a dynamically typed language like Python it is common to need to perform slightly different operation based on the type of an argument, especially when dealing with the difference between a list of items and a single item. It is simple enough to check the type of an argument directly, but in cases where the behavioral difference can be isolated into separate functions check the type of an argument directly, but in cases where the behavioral difference can be isolated into separate functions functools provides the singledispatch() decorator to register a set of *generic functions* for automatic switching based on the type of the first argument to a function.

```
# functools_singledispatch.py

import functools


@functools.singledispatch
def myfunc(arg):
    print('default myfunc({!r})'.format(arg))


@myfunc.register(int)
def myfunc_int(arg):
    print('myfunc_int({})'.format(arg))


@myfunc.register(list)
def myfunc_list(arg):
    print('myfunc_list()')
    for item in arg:
        print('  {}'.format(item))


myfunc('string argument')
myfunc(1)
myfunc(2.3)
myfunc(['a', 'b', 'c'])
```

The register() attribute of the new function serves as another decorator for registering alternative implementations. The first function wrapped with singledispatch() is the default implementation if no other type-specific function is found, as with the float case in this example.

```
$ python3 functools_singledispatch.py

default myfunc('string argument')
myfunc_int(1)
default myfunc(2.3)
myfunc_list()
  a
  b
  c
```

When no exact match is found for the type, the inheritance order is evaluated and the closest matching type is used.

```
# functools_singledispatch_mro.py

import functools


class A:
    pass


class B(A):
    pass


class C(A):
    pass


class D(B):
    pass


class E(C, D):
    pass
```

```
@functools.singledispatch
def myfunc(arg):
    print('default myfunc({})'.format(arg.__class__.__name__))


@myfunc.register(A)
def myfunc_A(arg):
    print('myfunc_A({})'.format(arg.__class__.__name__))


@myfunc.register(B)
def myfunc_B(arg):
    print('myfunc_B({})'.format(arg.__class__.__name__))


@myfunc.register(C)
def myfunc_C(arg):
    print('myfunc_C({})'.format(arg.__class__.__name__))


myfunc(A())
myfunc(B())
myfunc(C())
myfunc(D())
myfunc(E())
```

In this example, classes `D` and `E` do not match exactly with any registered generic functions, and the function selected depends on the class hierarchy.

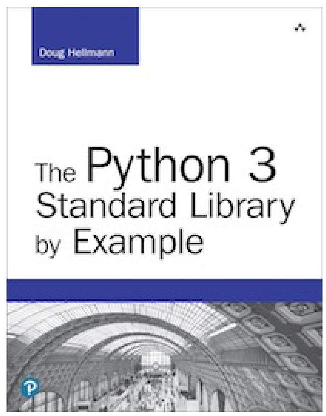
```
$ python3 functools_singledispatch_mro.py

myfunc_A(A)
myfunc_B(B)
myfunc_C(C)
myfunc_B(D)
myfunc_C(E)
```

See also

- Standard library documentation for functools
- Rich comparison methods – Description of the rich comparison methods from the Python Reference Guide.
- Isolated @memoize – Article on creating memoizing decorators that work well with unit tests, by Ned Batchelder.
- **PEP 443** – “Single-dispatch generic functions”
- inspect – Introspection API for live objects.

This page was last updated 2017-01-04.



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?