Iterators, Generators and Decorators

A Tutorial at EuroPython 2014

July 24, 2014

Berlin, Germany

author: Dr.-Ing. Mike Müller
email: mmueller@python-academy.de
twitter: @pyacademy

version: 2.0

url: http://www.python-academy.com/

Contents

Current Trainings Modules - Python Academy

As of August 2014

Module Topic	Length (days)	in-house	open
Python for Programmers	3	yes	yes
Python for Non-Programmers	4	yes	yes
Python for Programmers in Italian	3	yes	yes
Advanced Python	3	yes	yes
Introduction to Django	3	yes	yes
Advanced Django	3	yes	yes
Python for Scientists and Engineers	3	yes	yes
Fast Code with the Cython Compiler and Fast NumPy Processing with Cython	3	yes	yes
Professional Testing with pytest and tox	3	yes	yes
Twisted	3	yes	yes
Plone	2	yes	yes
Introduction to wxPython	2	yes	yes
Introduction to PySide/PyQt	2	yes	yes
SQLAlchemy	1	yes	yes
High Performance XML with Python	1	yes	yes
Camelot	1	yes	yes
Optimizing Python Programs	1	yes	yes
Python Extensions with Other Languages	1	yes	no
Data Storage with Python	1	yes	no
Introduction to Software Engineering with Python	1	yes	no
Overview of the Python Standard Library	1	yes	no
Threads and Processes in Python	1	yes	no
Windows Programming with Python	1	yes	no
Network Programming with Python	1	yes	no
Introduction to IronPython	1	yes	no

We offer on-site and open course all over Europe. We always customize and extend training modules as needed. We also provide consulting services such as code review, custom programming and tailor-made workshops.

More Information: www.python-academy.com

1 Iterators and Generators

1.1 Iterators

Iterators are objects that have a `next` and `__iter__` methods:

```
>>> class CountDown(object):
...     def __init__(self, start):
...         self.counter = start + 1
...     def next(self): # __next__ in Python 3
...         self.counter -= 1
...         if self.counter <= 0:
...             raise StopIteration
...         return self.counter
...     def __iter__(self):
...         return self
```

`__iter__` has to return the iterator itself and `next` should return the next element and raise `StopIteration` when finished. Now we can use our iterator:

```
>>> cd = CountDown(5)
>>> for x in cd:
...     print(x)
...
5
4
3
2
1
```

A sequence can be turned into an iterator using the built-in function `iter`:

```
>>> i = iter(range(5, 0, -1))
>>> next(i)
5
>>> next(i)
4
>>> i.next() # old way in Python 2 onlly
3
>>> next(i)
2
>>> next(i)
1
>>> next(i)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

1.2 Generator Functions

We can generate iterators with the help of generators. A generator function is a function with the keyword `yield` in its body:

```
>>> def count_down(value):
...     for x in xrange(value, 0, -1):
...         yield x
...
>>> c = count_down(5)
>>> next(c)
5
>>> next(c)
4
>>> next(c)
3
>>> next(c)
2
>>> next(c)
1
>>> next(c)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

1.3 Generator Expressions

Generator expressions look just like list comprehensions but don't use square brackets:

```
>>> exp = (x for x in xrange(5, 0, -1))
>>> for x in exp:
...     print(x)
...
5
4
3
2
1
```

1.4 Coroutines

The `yield` statement also excepts values that can be sent to the generator. Therefore, we call it a coroutine:

```
>>> def show_upper():
...     while True:
...         text = yield
...         print(text.upper())
...
```

After making an instance of it:

```
>>> s = show_upper()
```

we can send something to it:

```
>>> s.send('Hello')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

Well, almost. We need to call `next` at least once in order to get to yield where we can send something into the coroutine:

```
>>> s = show_upper()
>>> next(s)
>>> s.send('Hello')
HELLO
>>> s.send('there')
THERE
```

1.4.1 Automatic call to next

Since this is common thing to do, we can use a decorator that takes care of the first call to `next`:

```
>>> def init_coroutine(func):
...     def init(*args, **kwargs):
...         gen = func(*args, **kwargs)
...         next(gen)
...         return gen
...     return init
```

Now we can decorate our definition of the coroutine:

```
>>> @init_coroutine
... def show_upper():
...     while True:
...         text = yield
...         print(text.upper())
```

and can start sending without the call of `next`:

```
>>> s = show_upper()
>>> s.send('Hello')
HELLO
```

1.4.2 Sending and yielding at the same time

In addition to sending values, we can also receive some from the coroutine:

```
>>> @init_coroutine
... def show_upper():
...     result = None
...     while True:
...         text = yield result
...         result = text.upper()
...
>>> s = show_upper()
>>> res = s.send('Hello')
>>> res
'HELLO'
```

1.4.3 Closing a generator and raising exceptions

We can close the coroutine using the method `close`:

```
>>> s.close()
```

and will get an `StopIteration` exception:

```
>>> res = s.send('Hello')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

Calling `close` will throw a `GeneratorExit` exception inside the coroutine:

```
@init_coroutine
... def show_upper():
...     result = None
...     try:
...         while True:
...             text = yield result
...             result = text.upper()
...     except GeneratorExit:
...         print('done generating')
...
>>> s = show_upper()
>>> res = s.send('Hello')
>>> s.close()
done generating
```

Even if we catch this exception inside the coroutine, it will be closed:

```
>>> res = s.send('Hello')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
StopIteration
```

We can also raise an exception inside the coroutine from outside:

```
>>> s = show_upper()
>>> s.throw(NameError, 'Not known')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 6, in show_upper
NameError: Not known
```

Note that line 6 is the line in the coroutine with `yield`.

Analogous to files that are closed when the get out of scope, a `GeneratorExit` is raised when the object is garbage collected:

```
>>> s = show_upper()
>>> del s
done generating
```

1.5 Pipelining

Generators can be used to pipeline commands similar to UNIX shell commands. We have a program that generates a log file:

```
"""Creating a log files that is continuously updated.
"""

from __future__ import print_function

import random
import time


def log(file_name):
    """Write some random log data.
    """
    fobj = open(file_name, 'w')
    while True:
        value = random.randrange(0, 100)
        if value < 10:
            fobj.write('# comment\n')
        else:
            fobj.write('%d\n' % value)
        fobj.flush()
```

```
        time.sleep(2)

if __name__ == '__main__':

    def test():
        """Start logging.
        """
        import sys
        file_name = sys.argv[1]
        print('logging to', file_name)
        log(file_name)
    test()
```

The log file looks like this:

```
35
29
75
36
28
54
# comment
54
56
```

Now we can write a program with generators. We read the file and wait if there are currently no more new lines until new ones are written:

```
def read_forever(fobj):
    """Read from a file as long as there are lines.
    Wait for the other process to write more lines.
    """
    counter = 0
    while True:
        if counter > LIMIT:
            break
        line = fobj.readline()
        if not line:
            time.sleep(0.1)
            continue
        yield line
```

Then we filter out all comment lines:

```
def filter_comments(lines):
    """Filter out all lines starting with #.
    """
    for line in lines:
```



```
        if not line.strip().startswith('#'):
            yield line
```

and we convert the entry in the line into an integer:

```
def get_number(lines):
    """Read the number in the line and convert it to an integer.
    """
    for line in lines:
        yield int(line.split()[-1])
```

Finally, we pipe all these together and calculate the sum of all numbers and print it on the screen:

```
def show_sum(file_name='out.txt'):
    """Start all the generators and calculate the sum continuously.
    """
    lines = read_forever(open(file_name))
    filtered_lines = filter_comments(lines)
    numbers = get_number(filtered_lines)
    sum_ = 0
    try:
        for number in numbers:
            sum_ += number
            sys.stdout.write('sum: %d\r' % sum_)
            sys.stdout.flush()
    except KeyboardInterrupt:
        print('sum:', sum_)

if __name__ == '__main__':
    import sys
    show_sum(sys.argv[1])
```

1.6 Pipelining with Coroutines

While generators establish a pull pipeline, coroutines can create a push pipeline. Let's modify our log generator to include log levels:

```
"""Creating a log files that is continuously updated.

Modified version with log levels.
"""

from __future__ import print_function

import random
import time
```

```
LEVELS = ['CRITICAL', 'DEBUG', 'ERROR', 'FATAL', 'WARN']

def log(file_name):
    """Write some random log data.
    """
    fobj = open(file_name, 'w')
    while True:
        value = random.randrange(0, 100)
        if value < 10:
            fobj.write('# comment\n')
        else:
            fobj.write('%s: %d\n' % (random.choice(LEVELS), value))
        fobj.flush()
        time.sleep(2)

if __name__ == '__main__':

    def test():
        """Start logging.
        """
        import sys
        file_name = sys.argv[1]
        print('logging to', file_name)
        log(file_name)
    test()
```

Now, the log file looks like this:

```
ERROR: 78
DEBUG: 72
WARN: 99
CRITICAL: 97
FATAL: 40
FATAL: 33
CRITICAL: 34
ERROR: 18
ERROR: 89
ERROR: 46
FATAL: 49
WARN: 95
```

We use our decorator to advance a coroutine to the first `yield`:

```
"""Use coroutines to sum log file data with different log levels.
"""

import functools
import sys
```



```python
import time

LIMIT = 1000000

def init_coroutine(func):
    @functools.wraps(func)
    def init(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return init
```

The function for reading the file line-by-line takes the argument `target`. This is a coroutine that will consume the line:

```python
def read_forever(fobj, target):
    """Read from a file as long as there are lines.
    Wait for the other process to write more lines.
    Send the lines to `target`.
    """
    counter = 0
    while True:
        if counter > LIMIT:
            break
        line = fobj.readline()
        if not line:
            time.sleep(0.1)
            continue
        target.send(line)
```

We have two coroutines that receive values with `line = yield` and send their their computed results to `target`:

```python
@init_coroutine
def filter_comments(target):
    """Filter out all lines starting with #.
    """
    while True:
        line = yield
        if not line.strip().startswith('#'):
            target.send(line)


@init_coroutine
def get_number(targets):
    """Read the number in the line and convert it to an integer.
    Use the level read from the line to choose the to target.
    """
```

```
    while True:
        line = yield
        level, number = line.split(':')
        number = int(number)
        targets[level].send(number)
```

We define a consumer for each logging level:

```
# Consumers for different cases.

@init_coroutine
def fatal():
    """Handle fatal errors."""
    sum_ = 0
    while True:
        value = yield
        sum_ += value
        sys.stdout.write('FATAL    sum: %7d\n' % sum_)
        sys.stdout.flush()


@init_coroutine
def critical():
    """Handle critical errors."""
    sum_ = 0
    while True:
        value = yield
        sum_ += value
        sys.stdout.write('CRITICAL sum: %7d\n' % sum_)


@init_coroutine
def error():
    """Handle normal errors."""
    sum_ = 0
    while True:
        value = yield
        sum_ += value
        sys.stdout.write('ERROR    sum: %7d\n' % sum_)


@init_coroutine
def warn():
    """Handle warnings."""
    sum_ = 0
    while True:
        value = yield
        sum_ += value
        sys.stdout.write('WARN     sum: %7d\n' % sum_)
```

```
@init_coroutine
def debug():
    """Handle debug messages."""
    sum_ = 0
    while True:
        value = (yield)
        sum_ += value
        sys.stdout.write('DEBUG    sum: %7d\n' % sum_)
```

and collect the coroutines in a dictionary:

```
TARGETS = {'CRITICAL': critical(),
           'DEBUG': debug(),
           'ERROR': error(),
           'FATAL': fatal(),
           'WARN': warn()}
```

Now we can start pushing the data through our coroutine pipeline:

```
def show_sum(file_name='out.txt'):
    """Start start the pipline.
    """
    # read_forever > filter_comments > get_number > TARGETS
    read_forever(open(file_name), filter_comments(get_number(TARGETS)))


if __name__ == '__main__':
    show_sum(sys.argv[1])
```

The resulting output will look like this:

```
FATAL    sum:      80
ERROR    sum:      68
FATAL    sum:     178
CRITICAL sum:      23
DEBUG    sum:      27
CRITICAL sum:      91
CRITICAL sum:     125
FATAL    sum:     230
CRITICAL sum:     223
CRITICAL sum:     260
```

1.7 Itertools

The `itertools` module in the standard library offers powerful functions that work with and return iterators.

We import the module:

```
>>> import itertools as it
```

We can have an infinity iterator that starts at the beginning after reaching its end with `cycle`:

```
>>> cycler = it.cycle([1,2,3])
>>> next(cycler)
1
>>> next(cycler)
2
>>> next(cycler)
3
>>> next(cycler)
1
>>> next(cycler)
2
>>> next(cycler)
3
>>> next(cycler)
1
```

The function `counter` provides an infinite counter with an optional start value (default is zero):

```
>>> counter = it.count(5)
>>> next(counter)
5
>>> next(counter)
6
```

With `repeat` we can construct a new iterator that also can be infinite:

```
>>> list(it.repeat(4, 2))
[4, 4]
>>> list(it.repeat(2, 4))
[2, 2, 2, 2]
>>> endless  = it.repeat(3)
>>> next(endless)
3
>>> next(endless)
3
>>> next(endless)
3
```

We can use `iszip` to zip two or more iterables:

```
>>> list(it.izip([1,2,3], [4,5,6,7,8]))
[(1, 4), (2, 5), (3, 6)]
```

The variation `izip_longest` fills missing values:

```
>>> list(it.izip_longest([1,2,3], [4,5,6,7,8]))
[(1, 4), (2, 5), (3, 6), (None, 7), (None, 8)]
>>> list(it.izip_longest([1,2,3], [4,5,6,7,8], fillvalue=99999))
[(1, 4), (2, 5), (3, 6), (99999, 7), (99999, 8)]
```

Two or more iterables can be combine in one with `chain`:

```
>>> list(it.chain([1,2,3], [4,5,6]))
[1, 2, 3, 4, 5, 6]
```

To get only part of an iterable, we can use `islice` that works very similar to the slicing of sequences:

```
>>> list(it.islice(range(10), 5))
[0, 1, 2, 3, 4]
>>> list(it.islice(range(10), 5, 8))
[5, 6, 7]
>>> list(it.islice(range(10), 5, None))
[5, 6, 7, 8, 9]
>>> list(it.islice(range(10), 5, None, 2))
[5, 7, 9]
>>>
```

1.8 Exercises

1. Write a generator that creates an endless stream of numbers starting from a value given as argument with a step size of 5. Write one version without and one with `itertools`.
2. Extend this generator into an coroutine that allows the step size to be set from outside.
3. Stop the coroutine after it has produced 10 values (a) form outside and (b) from inside the coroutine.
4. Rewrite the following code snippets using `itertools`.

```
x = 0
while True:
    x += 1

[1, 2, 3, 4, 5][2:]
[1, 2, 3, 4, 5][:4]

[1, 2, 3] + [4, 5, 6]

zip('abc', [1, 2, 3])
```


2 Decorators

2.1 The Origin

Decorator provide a very useful method to add functionality to existing functions and classes. Decorators are functions that wrap other functions or classes.

One example for the use of decorator are static methods. Static methods could be function in the global scope but are defined inside a class. There is no``self`` and no reference to the instance. Before Python 2.4 they had to defined like this:

```
>>> class C(object):
...     def func():
...         """No self here."""
...         print('Method used as function.')
...     func = staticmethod(func)
...
>>> c = C()
>>> c.func()
Method used as function.
```

Because the `staticmethod` call is after the actual definition of the method, it can be difficult to read an easy to be overlooked. Therefore, the new `@` syntax is used before the method definition but does the same:

```
>>> class C(object):
...     @staticmethod
...     def func():
...         """No self here."""
...         print('Method used as function.')
...
>>> c = C()
>>> c.func()
Method used as function.
```

The same works for class methods that take a class objects as argument instead of the instance (aka `self`).

2.2 Write Your Own

Writing you own decorator is simple:

```
>>> def hello(func):
...     print('Hello')
...
```

Now apply it a function:

```
>>> @hello
... def add(a, b):
```

```
...     return a + b
...
Hello
```

The `Hello` got printed. But calling our `add` doesn't work:

```
>>> add(10, 20)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

This might become clearer if look at it the old way:

```
>>> def add(a, b):
...     return a + b
...
>>> add = hello(add) # hello has no return value, i.e None
Hello
>>> add
>>> add(20, 30)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

So, even it is not enforced by the interpreter, decorators usually make sense (at least the way they are intended to be use) if they behave in a certain way. It is strongly recommended that a function decorator always returns a function object and a class decorator always returns a class object. A function decorator should typically either return a function that returns the result of the call to the original function and do something in addition or return the original function itself.

This is a more useful example:

```
>>> def hello(func):
        """Decorator function."""
...     def call_func(*args, **kwargs):
            """Takes a arbitrary number of positional and keyword arguments."""
...         print('Hello')
...         # Call original function and return its result.
...         return func(*args, **kwargs)
...     # Return function defined in this scope.
...     return call_func
```

Now we can create our decorated function and call it:

```
>>> @hello
... def add(a, b):
...     return a + b
...
```

```
>>> add(20, 30)
Hello
50
```

and again:

```
>>> add(20, 300)
Hello
320
```

2.3 Parameterized Decorators

Decorators can take arguments. We redefine our decorator. The outermost function takes the arguments, the next more inner function takes the function and the innermost function will be returned and will replace the original function:

```
>> def say(text):
...     def _say(func):
...         def call_func(*args, **kwargs):
...             print(text)
...             return func(*args, **kwargs)
...         return call_func
...     return _say
```

Now we decorate with `Hello` to get the same effect as before:

```
>>> @say('Hello')
... def add(a, b):
...     return a, b
...
>>> add(10, 20)
Hello
(10, 20)
```

or with `Goodbye`:

```
>>> @say('Goodbye')
... def add(a, b):
...     return a, b
...
>>> add(10, 20)
Goodbye
(10, 20)
```

2.4 Chaining Decorators

We can use more than one decorator for one function:

```
>>> @say('A')
... @say('B')
... @hello
... def add(a, b):
...     return a, b
...
>>> add(10, 20)
A
B
Hello
(10, 20)
```

2.5 Class Decorators

Since Python 2.6 we can use decorators for classes too:

```
>>> def mark(cls):
...     cls.added_attr = 'I am decorated.'
...     return cls
...
>>> @mark
... class A(object):
...     pass
...
>>> A.added_attr
'I am decorated.'
```

It is important to always return a class object from the decorating function. Otherwise users cannot make instances from our class.

2.6 Best Practice

When we use docstrings, as we always do:

```
>>> def add(a, b):
...     """Add two objects."""
...     return a + b
```

we can access them later:

```
>>> add.__doc__
'Add two objects.'
```

When we now wrap our function:

```
>>> def hello(func):
...     def call_func(*args, **kwargs):
```

```
...         """Wrapper."""
...         print('Hello')
...         return func(*args, **kwargs)
...     return call_func
...
```

and decorate it:

```
>>> @hello
... def add(a, b):
...     """Add two objects."""
...     return a, b
```

we loose our docstring:

```
>>> add.__doc__
'Wrapper.'
```

We could manually set the docstring of our wrapped function to remain the same. But the module `functools` in the standard library helps us here:

```
>>> import functools
>>> def hello(func):
...     @functools.wraps(func)
...     def call_func(*args, **kwargs):
...         """Wrapper."""
...         print('Hello')
...         return func(*args, **kwargs)
...     return call_func
...
```

Now we have a nice docstring after decorating our function:

```
>>> @hello
... def add(a, b):
...     """Add two objects."""
...     return a + b
...
>>> add.__doc__
'Add two objects.'
```

Python allows to call function recursively:

```
>>> def recurse(x):
...     if x:
...         x -= 1
...         print(x)
```

```
...         recurse(x)
...
>>> recurse(5)
4
3
2
1
0
```

When we decorate a recursive function the wrapper will also be called recursively:

```
>>> @hello
... def recurse(x):
...     if x:
...         x -= 1
...         print(x)
...         recurse(x)
...
>>> recurse(5)
Hello
4
Hello
3
Hello
2
Hello
1
Hello
0
Hello
```

I most cases this is not desirable. Therefore, recursive function should not be decorated. Don't assume you have only one decorator.

2.7 Use cases

Decorators can be used for different purposes some common one are shown below.

2.7.1 Argument Checking

We check if the positional arguments to a function call are of a certain type. First we define our decorator:

```
"""Check function arguments for given type.
"""

import functools
```

```
def check(*argtypes):
    """Function argument type checker.
    """

    def _check(func):
        """Takes the function.
        """

        @functools.wraps(func)
        def __check(*args):
            """Takes the arguments
            """
            if len(args) != len(argtypes):
                msg = 'Expected %d but got %d arguments' % (len(argtypes),
                                                             len(args))
                raise TypeError(msg)
            for arg, argtype in zip(args, argtypes):
                if not isinstance(arg, argtype):
                    msg = 'Expected %s but got %s' % (
                        argtypes, tuple(type(arg) for arg in args))
                    raise TypeError(msg)
            return func(*args)
        return __check
    return _check
```

Then we decorate our function:

```
>>> from argcheck import check
>>> @check(int, int)
... def add(x, y):
...     """Add two integers."""
...     return x + y
```

We have our docstring:

```
>>> add.__doc__
'Add two integers.'
```

and can call it with two integers:

```
>>> add(1, 2)
3
```

But calling with an integer and a float doesn't work:

```
>>> add(1, 2.0)
Traceback (most recent call last):
```

```
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 11, in __check
TypeError: Expected (<type 'int'>, <type 'int'>) but got (<type 'int'>, <type 'float'>)
```

Also the wrong number of parameters won't work:

```
>>> add(1)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 7, in __check
TypeError: Expected 2 but got 1 arguments
>>> add(1,1,1)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 7, in __check
TypeError: Expected 2 but got 3 arguments
```

We can't use our function if we have a different number of parameters in the decorator than in the function definition:

```
>>> @check(int, int, int)
... def add(x, y):
...     """Add two integers."""
...     return x + y
...
>>> add(1, 2)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 7, in __check
TypeError: Expected 3 but got 2 arguments
```

2.7.2 Caching

Expensive but repeated calculations can be cached. A simple cache for a function never expires and grows without limit could like this:

```
"""Caching results with a decorator.
"""

import functools
import pickle


def cached(func):
    """Decorator that caches.
    """
    cache = {}
```

```
    @functools.wraps(func)
    def _cached(*args, **kwargs):
        """Takes the arguments.
        """
        # dicts cannot be use as dict keys
        # dumps are strings and can be used
        key = pickle.dumps((args, kwargs))
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]
    return _cached
```

Now we can decorated our expensive function:

```
>>> from cached import cached
>>> @cached
... def add(a, b):
...     print('calc')
...     return a + b
```

Only the first call will print `calc`. All subsequent calls get the value from cache without newly calculating it:

```
>>> add(10, 10)
calc
20
>>> add(10, 10)
20
>>> add(10, 10)
20
```

2.7.3 Logging

Another use case is logging. We log things if the global variable `LOGGING` is true:

```
"""Helper to switch on and off logging of decorated functions.
"""

from __future__ import print_function

import functools

LOGGING = False


def logged(func):
    """Decorator for logging.
    """
```

```
    @functools.wraps(func)
    def _logged(*args, **kwargs):
        """Takes the arguments
        """
        if LOGGING:
            print('logged') # do proper logging here
        return func(*args, **kwargs)
    return _logged
```

After decorating our function:

```
>>> import logged
>>> @logged.logged
... def add(a, b):
...     return a + b
```

an setting `LOGGING` to true:

```
>>> logged.LOGGING = True
```

we log:

```
>>> add(10, 10)
logged
20
```

or not:

```
>>> logged.LOGGING = False
>>> add(10, 10)
20
```

2.7.4 Registration

Another useful application is registration. We would like to register functions. The first way is to make them append themselves to a list when they are called. We use a dictionary `registry` to store these lists. This is our decorator:

```
"""A function registry.
"""

import functools

registry = {}


def register_at_call(name):
```

```
    """Register the decorated function at call time.
    """

    def _register(func):
        """Takes the function.
        """

        @functools.wraps(func)
        def __register(*args, **kwargs):
            """Takes the arguments.
            """
            registry.setdefault(name, []).append(func)
            return func(*args, **kwargs)
        return __register
    return _register
```

and this our empty registry:

```
>>> from registering import registry, register_at_call
>>> registry
{}
```

We define three decorated functions:

```
>>> @register_at_call('simple')
... def f1():
...     pass
...
>>> @register_at_call('simple')
... def f2():
...     pass
...
>>> @register_at_call('complicated')
... def f3():
...     pass
```

The registry is still empty:

```
>>> registry
{}
```

Now we call our functions and fill the registry:

```
>>> f1()
>>> registry
{'simple': [<function f1 at 0x00F97730>]}
>>> f2()
```

```
>>> registry
{'simple': [<function f1 at 0x00F97730>,
            <function f2 at 0x00F97B70>]}
>>> f3()
>>> registry
{'simple': [<function f1 at 0x00F97730>,
            <function f2 at 0x00F97B70>],
'complicated': [<function f3 at 0x00F976F0>]}
```

We can also look at the names of our functions:

```
>>> f1.__name__
'f1'
>>> [f.__name__ for f in registry['simple']]
['f1', 'f2']
>>> [f.__name__ for f in registry['complicated']]
['f3']
```

Of course we will append a function every time we call it:

```
>>> registry
{'simple': [<function f1 at 0x00F97730>,
            <function f2 at 0x00F97B70>,
            <function f1 at 0x00F97730>],
'complicated': [<function f3 at 0x00F976F0>]}
```

If want to register our function at definition time, we have to change our decorator:

```
def register_at_def(name):
    """Register the decorated function at definition time.
    """

    def _register(func):
        """Takes the function.
        """
        registry.setdefault(name, []).append(func)

        return func
    return _register
```

Now we add our function right when we define it:

```
>>> from registering import register_at_def
>>> registry = {}
>>> @register_at_def('simple')
... def f1():
...     pass
```

```
...
>>> registry
{'simple': [<function f1 at 0x00F97C70>]}
>>>
```

Calling doesn't change anything in the registry:

```
>>> f1()
>>> registry
{'simple': [<function f1 at 0x00F97C70>]}
>>> f1()
>>> registry
{'simple': [<function f1 at 0x00F97C70>]}
```

2.7.5 Verification

Verification is another useful way to use decorators. Lets make sure we have fluid water:

```
>>> def assert_fluid(cls):
...     assert 0 <= cls.temperature <= 100
...     return cls
```

We decorate our class:

```
>>> @assert_fluid
... class Water(object):
...     temperature = 20
```

It won't work if it is too hot or too cold:

```
>>> @assert_fluid
... class Steam(object):
...     temperature = 120
...
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
  File "<interactive input>", line 2, in assert_fluid
AssertionError
>>> @assert_fluid
... class Ice(object):
...     temperature = -20
...
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
  File "<interactive input>", line 2, in assert_fluid
AssertionError
```

2.8 Exercises

1. Write a function decorator that can be used to measure the run time of a functions. Use `timeit.default_timer` to get time stamps.
2. Make the decorator parameterized. It should take an integer that specifies how often the function has to be run. Make sure you divide the resulting run time by this number.
3. Use `functools.wraps` to preserve the function attributes including the docstring that you wrote.
4. Make the time measurement optional by using a global switch in the module that can be set to `True` or `False` to turn time measurement on or off.
5. Write another decorator that can be used with a class and registers every class that it decorates in a dictionary.

3 About Python Academy

Current Trainings Modules - Python Academy

As of August 2014

Module Topic	Length (days)	in-house	open
Python for Programmers	3	yes	yes
Python for Non-Programmers	4	yes	yes
Python for Programmers in Italian	3	yes	yes
Advanced Python	3	yes	yes
Introduction to Django	3	yes	yes
Advanced Django	3	yes	yes
Python for Scientists and Engineers	3	yes	yes
Fast Code with the Cython Compiler and Fast NumPy Processing with Cython	3	yes	yes
Professional Testing with pytest and tox	3	yes	yes
Twisted	3	yes	yes
Plone	2	yes	yes
Introduction to wxPython	2	yes	yes
Introduction to PySide/PyQt	2	yes	yes
SQLAlchemy	1	yes	yes
High Performance XML with Python	1	yes	yes
Camelot	1	yes	yes
Optimizing Python Programs	1	yes	yes
Python Extensions with Other Languages	1	yes	no
Data Storage with Python	1	yes	no
Introduction to Software Engineering with Python	1	yes	no
Overview of the Python Standard Library	1	yes	no
Threads and Processes in Python	1	yes	no
Windows Programming with Python	1	yes	no
Network Programming with Python	1	yes	no
Introduction to IronPython	1	yes	no

We offer on-site and open course all over Europe. We always customize and extend training modules as needed. We also provide consulting services such as code review, custom programming and tailor-made workshops.

More Information: www.python-academy.com