Advanced Python

generators, decorators, context managers

Zbigniew Jędrzejewski-Szmek

George Mason University

Python Summer School, Zürich, September 05, 2013

Version `Zürich-98-ge13be00`

This work is licensed under the *Creative Commons CC BY-SA 3.0 License.*

Outline

Subroutines

https://en.wikipedia.org/wiki/Subroutine#Advantages

The advantages of breaking a program into subroutines include:

- decomposing a complex programming task into simpler steps
- reducing duplicate code within a program
- enabling reuse of code across multiple programs
- dividing a large programming task among various programmers
- hiding implementation details from users of the subroutine improving traceability

Extracting subroutine wrappers

```
def subroutine1(arg1, arg2):
    log.debug('entering subroutine1 (%s)' % (arg1, arg2))
    ...
    log.debug('returning from subroutein1')

def subroutine2(arg):
    log.debug('entering sub2 (%s)' % arg)
    ...
    log.debug('returning from sub2')
```

```
@log_entrance
def subroutine1(arg1, arg2):
    ...

@log_entrance
def subroutine2(arg):
    ...
```

Extracting subroutine error handling

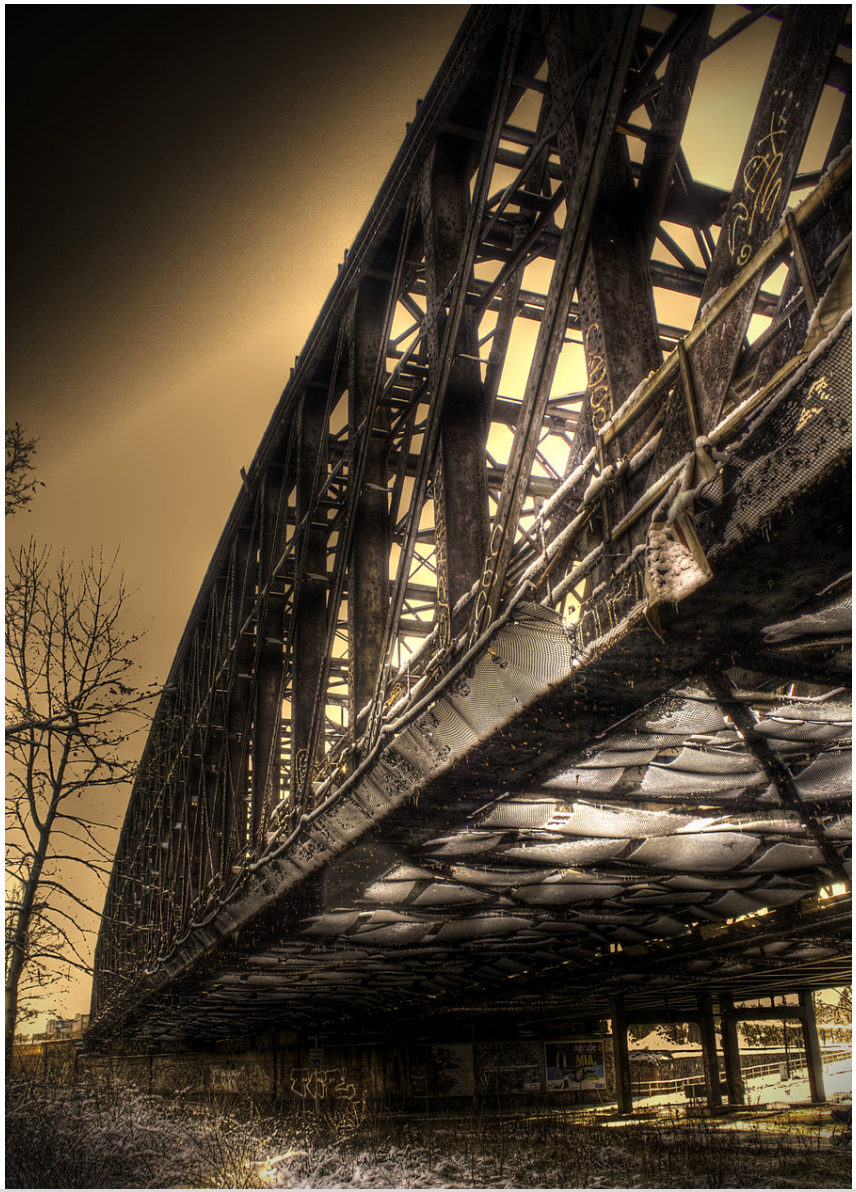
```
database_connect()
try:
    ...
finally:
    database_disconnect()

create_temporary_directory()
try:
    database_connect()
    try:
        ...
    finally:
        database_disconnect()
finally:
    remove_temporary_directory()
```

```
with database_lock():
    ...

with (TemporaryDirectory(),
      database_lock()):
    ...
```

Decorators



Decorators

Summary

This amazing feature appeared in the language almost apologetically and with concern that it might not be that useful.

Bruce Eckel

- decorators?
  passing of a function object through a filter + syntax
- can *work* on classes or functions
- can be *written* as classes or functions
- nothing new under the sun ;)
  - function could be written differently
  - syntax equivalent to explicit decorating function call and assignment
  - just cleaner

Syntax

```
@deco
def func():
    print('in func')
```

```
def func():
    print('in func')
func = deco(func)
```

```
def deco(orig_f):
    print('decorating:', orig_f)
    return orig_f
```

A decorator doing something...

register a function

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

```
# old style
...
def hello():
    return "Hello World!"
hello = app.route("/")(hello)
...
```

. . . written as a class

register a function

```
TABLE = {}

class register(object):
    def __init__(self, name):
        self.name = name
    def __call__(self, function):
        TABLE[name] = function
        return function
```

. . . written as as nested functions

register a function

```
TABLE = {}

def register(name):
    def helper(orig_f):
        TABLE[name] = function
        return orig_f
    return helper
```

Replace a function

```
class deprecated(object):
    "Print a deprecation warning"
    def __init__(self):
        pass
    def __call__(self, func):
        self.func = func
        return self.wrapper
    def wrapper(self, *args, **kwargs):
        print(self.func.__name__, 'is deprecated')
        return self.func(*args, **kwargs)
```

```
>>> @deprecated()
... def f(): pass
>>> f()
f is deprecated
```

Replace a function

alternate version

```
class deprecated(object):
    "Print a deprecation warning once"
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        print(self.func.__name__, 'is deprecated')
        return self.func(*args, **kwargs)
```

```
>>> @deprecated
... def f(): pass
>>> f()
f is deprecated
```

Decorators can be stacked

```
@app.route("/esp")
@log_entry
def func():
    return "Hola mundo"


# old style
def func():
    return "Hola mundo"
func = app.route("/esp")(log_entry(func))
```

The docstring problem

Our beautiful replacement function is missing

- the docstring
- attributes
- proper argument list

```
functools.update_wrapper(wrapper, wrapped)
```

- `__doc__`
- `__module__` and `__name__`
- `__dict__`
- `eval` is required for the rest :(
- module `decorator` compiles functions dynamically

Replace a function, keep the docstring

```
import functools

def deprecated(func):
    """Print a deprecation warning once"""
    func.count = 0
    def wrapper(*args, **kwargs):
        func.count += 1
        if func.count == 1:
            print(func.__name__, 'is deprecated')
        return func(*args, **kwargs)
    return functools.update_wrapper(wrapper, func)
```

pickling!

Decorators work for classes too

- same principle
- much less exciting

```
@deco
class A(object):
    pass
```

Decorators for methods

```
class A(object):
    def method(self, *args):
        return 1

    @classmethod
    def cmethod(cls, *args):
        return 2


    @staticmethod
    def smethod(*args):
       return 3

    @property
    def notamethod(*args):
        return 4
```

```
>>> a = A()
>>> a.method()
1

>>> a.cmethod()
2
>>> A.cmethod()
2

>>> a.smethod()
3
>>> A.smethod()
3

>>> a.notamethod
4
```

The property decorator

```
class Square(object):
    def __init__(self, edge):
        self.edge = edge

    @property
    def area(self):
        """Computed area.
          """
        return self.edge**2
```

```
>>> Square(2).area
4
```

The property decorator

```
class Square(object):
    def __init__(self, edge):
        self.edge = edge

    @property
    def area(self):
        """Computed area.
          Setting this updates the edge length!
          """
        return self.edge**2

    @area.setter
    def area(self, area):
        self.edge = area ** 0.5
```

The property triple: setter, getter, deleter

- attribute access `a.edge` calls `area.getx`
  - set with `@property`
- attribute setting `a.edge=3` calls `area.setx`
  - set with `.setter`
- attribute setting `del a.edge` calls `area.delx`
  - set with `.deleter`

Summary (decorators)

A decorator can be used to:

- do stuff when a function or class is created

or

- replace a function or a class and modify behaviour completely

Resource cleanup

Exception handling

```
try:
    return 1/0
except ZeroDivisionError as description:
    print('got', description)
    return float('inf')
```

Freeing stuff in `finally`

How to make sure resources are freed?

```
resource = open(...)
try:
    do_something(resource)
finally:
    resource.close()
```

Using a context manager

```
with manager as var:
    do_something(var)
```

```
var = manager.__enter__()
try:
    do_something(var)
finally:
    manager.__exit__(None, None, None)
```

Context manager: closing

```
class closing(object):
    def __init__(self, obj):
        self.obj = obj
    def __enter__(self):
        return self.obj
    def __exit__(self, *args):
        self.obj.close()
```

```
>>> with closing(open('/tmp/file', 'w')) as f:
...   f.write('the contents\n')
```

file is a context manager

```
>>> help(file.__enter__)
Help on method_descriptor:

__enter__(...)
    __enter__() -> self.
>>> help(file.__exit__)
Help on method_descriptor:

__exit__(...)
    __exit__(*excinfo) -> None.  Closes the file.

>>> with open('/tmp/file', 'a') as f:
...   f.write('the contents\n')
```

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`
- `decimal.localcontext`
- `warnings.catch_warnings`
- `contextlib.closing`
- parallel programming
  - `concurrent.futures.ThreadPoolExecutor` (3.2)
  - `concurrent.futures.ProcessPoolExecutor` (3.2)
- testing
  - `unittest.TestCase.assertRaises` (2.7)

Generators

How can we create an iterator?

1. write a class with `.__iter__` and `.next`
2. use a generator expression
3. write a generator function

Generator functions

```
>>> def gen():
...     print('--start')
...     yield 1
...     print('--middle')
...     yield 2
...     print('--stop')
```

```
>>> g = gen()
>>> g.next()
--start
1
>>> g.next()
--middle
2
>>> g.next()
--stop
Traceback (most recent call last):
  ...
StopIteration
```

yield as an expression

```
def gen():
    val = yield
```

Some `value` is sent when `gen().send(value)` is used, not `gen().next()`

Sending information **to** the generator

```
def gen():
  print('--start')
  val = yield 1
  print('--got', val)
  print('--middle')
  val = yield 2
  print('--got', val)
  print('--done')
```

```
>>> g = gen()
>>> g.next()
--start
1
>>> g.send('boo')
--got boo
--middle
2
>>> g.send('foo')
--got foo
--done
Traceback (most recent call last):
  ...
StopIteration
```

Throwing exceptions **into** the generator

```
>>> def f():
...     try:
...         yield
...     except GeneratorExit:
...         print("bye!")
>>> g = ()
>>> g.next()
>>> g.throw(IndexError)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
IndexError
```

Destroying generators

`.close()` is used to destroy resources tied up in the generator

```
>>> def f():
...     try:
...         yield
...     except GeneratorExit:
...         print("bye!")

>>> g = f()
>>> g.next()
>>> g.close()
bye!
```

Writing context managers as generators

Writing context managers as generators

```
@contextlib.contextmanager
def some_generator(<arguments>):
    <setup>
    try:
        yield <value>
    finally:
        <cleanup>


class Manager(object):
    def __init__(self, <arguments>):
        ...
    def __enter__(self):
        <setup>
        return <value>
    def __exit__(self, *exc_info):
        <cleanup>
```

Context manager: closing

```
@contextlib.contextmanager
def closing(f):
    try:
        yield f
    finally:
        f.close()
```

Nested finallys

```
try:
    try:
        print('work')
        buggy_code()
    finally:
        print 'finalizer a'
        cleanup(1/0)
finally:
    print 'finalizer b'
```

```
work
finalizer a
finalizer b
Traceback (most recent call last
  ...
ZeroDivisionError: integer
 division or modulo by zero
```

Nested context managers

```
>>> @contextlib.contextmanager
... def cm(name):
...     print(name, 'enter')
...     yield
...     print(name, 'exit')

>>> with cm('outer'), cm('inner'):
...     print('----')
outer enter
inner enter
----
inner exit
outer exit
```

For completeness: `else`

another less well-known thing that can dangle after a try clause...

```
try:
    ans = math.sqrt(num)
except ValueError:
    print('operation failed, returning dummy value')
    ans = float('nan')
else:
    print('operation succeeded')
```

Unittesting thrown exceptions

```
def test_indexing():
    try:
        address_book.lookup('Personus Nonexistus')
    except KeyError:
        pass
```

Can we do better?

```
import py.test
def test_indexing():
    unittest.TestCase.assertRaises(KeyError,
                                   address_book.lookup,
                                   'Personus Nonexistus')
```

Can we do better?

Unittesting thrown exceptions

```
def test_indexing():
    with assertRaises(KeyError)
        address_book.lookup('Personus Nonexistus')
```

Managing exceptions

```
class Manager(object):
    ...
    def __exit__(self, type, value, traceback):
        ...
        return suppress
@contextlib.contextmanager
def assertRaises(exc):
    try:
        yield
    except exc:
        pass
    except Exception as value:
        raise AssertionError('wrong exception type')
    else:
        raise AssertionError(exc.__name__+' expected')
```

Summary

- Decorators make wrapping and altering functions and classes easy
- Generators make iterators easy
- Context managers make outsourcing `try...except..finally` blocks easy

Decorators and context managers make beautiful Python code

That's all!

Chaining generators

With `.send()` and `.throw()` chaining generators is complicated

yield from <subiterator> 3.3

```
def words(file):
    for line in file:
        for word in words(line):
            yield word
```

```
def words(file):
    for line in file:
        sub = words(line)
        yield from sub
```