Advanced Python

decorators, generators, context managers

Zbigniew Jędrzejewski-Szmek

Institute of Experimental Physics
University of Warsaw

Python Summer School Kiel, September 6, 2012

Version `AP_version_2011-382-gd5149a9`

Outline

Generators

Generator functions

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

Generator functions

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

```
>>> g = gen()
```

Generator functions

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

```
>>> g = gen()
>>> g.next()
--start
1
```

Generator functions

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

```
>>> g = gen()
>>> g.next()
--start
1
>>> g.next()
--middle
2
```

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
```

Simple generator function

```
def countdown(n):
    print "Counting down from", n
    while n > 0:
        yield n
        n -= 1
```

Simple generator function

```
def countdown(n):
    print "Counting down from", n
    while n > 0:
        yield n
        n -= 1

>>> list(countdown(10))
Counting down from 10
[10 9 8 7 6 5 4 3 2 1]
```

Iterate over words

```
def words(input):
    word = ''
    while True:
        c = input.read(1)
        if not c:
            break
        if c.isspace():
            if word:
                yield word
            word = ''
        else:
            word += c
    if word:
        yield word
```

yield as an expression

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

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'
```

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()
```

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
```

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
```

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():
...     yield
>>> g = f()
```

Throwing exceptions **into** the generator

```
>>> def f():
...     yield
>>> g = f()
>>> g.next()
```

Throwing exceptions **into** the generator

```
>>> def f():
...     yield
>>> g = f()
>>> 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

Destroying generators

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

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

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!
```

Chaining generators

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

Chaining generators

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

yield from <subiterator>

3.3

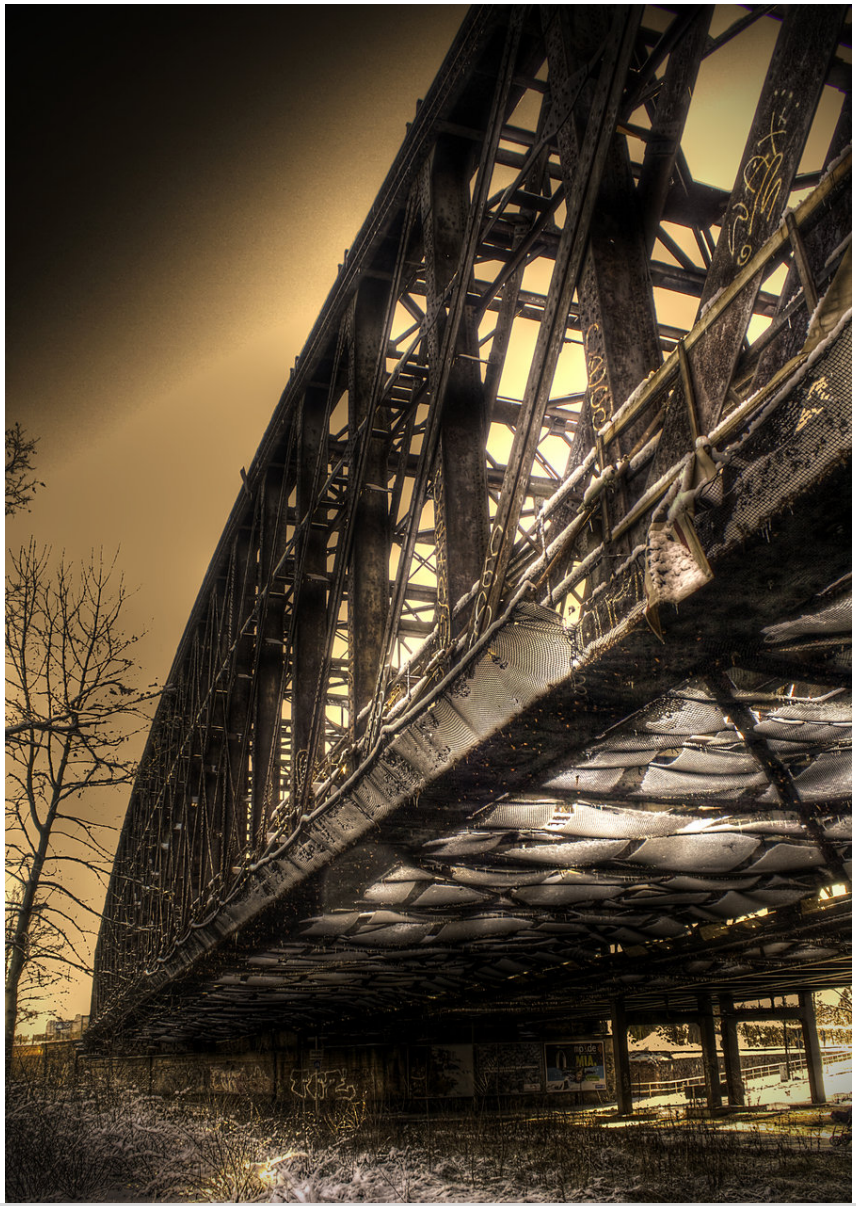
Chaining generators

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

yield from <subiterator> 3.3

```
def chain(*generators):
    for g in generators:
        yield from g
```

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?

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

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

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

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'
```

Syntax

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

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

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...

set an attribute on the function object

A decorator doing something...

set an attribute on the function object

```
>>> @author('Joe')
... def func(): pass
>>> func.author
'Joe'
```

A decorator doing something...

set an attribute on the function object

```
>>> @author('Joe')
... def func(): pass
>>> func.author
'Joe'
```

```
# old style
>>> def func(): pass
>>> func = author('Joe')(func)
>>> func.author
'Joe'
```

. . . written as a class

set an attribute on the function object

```
class author(object):
    def __init__(self, name):
        self.name = name
    def __call__(self, function):
        function.author = self.name
        return function
```

. . . written as as nested functions

set an attribute on the function object

```
def author(name):
    def helper(orig_f):
        orig_f.author = name
        return orig_f
    return helper
```

Replace a function

Replace a function

```
class deprecated(object):
    "Print a deprecation warning once"
    def __init__(self):
        pass
    def __call__(self, func):
        self.func = func
        self.count = 0
        return self.wrapper
    def wrapper(self, *args, **kwargs):
        self.count += 1
        if self.count == 1:
            print self.func.__name__, 'is deprecated'
        return self.func(*args, **kwargs)
```

Replace a function

```
class deprecated(object):
    "Print a deprecation warning once"
    def __init__(self):
        pass
    def __call__(self, func):
        self.func = func
        self.count = 0
        return self.wrapper
    def wrapper(self, *args, **kwargs):
        self.count += 1
        if self.count == 1:
            print self.func.__name__, 'is deprecated'
        return self.func(*args, **kwargs)
```

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

Replace a function

alternate version

```
class deprecated(object):
    "Print a deprecation warning once"
    def __init__(self, func):
        self.func = func
        self.count = 0
    def __call__(self, *args, **kwargs):
        self.count += 1
        if self.count == 1:
            print self.func.__name__, 'is deprecated'
        return self.func(*args, **kwargs)
```

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

Decorators can be stacked

```
@author('Joe')
@deprecated
def func():
    pass


# old style
def func():
    pass
func = author('Joe')(deprecated(func))
```

The docstring problem

Our beautiful replacement function lost

- the docstring
- attributes
- proper argument list

The docstring problem

Our beautiful replacement function lost

- the docstring
- attributes
- proper argument list

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

The docstring problem

Our beautiful replacement function lost

- the docstring
- attributes
- proper argument list

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

- `__doc__`

The docstring problem

Our beautiful replacement function lost

- the docstring
- attributes
- proper argument list

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

- `__doc__`
- `__module__` and `__name__`

The docstring problem

Our beautiful replacement function lost

- the docstring
- attributes
- proper argument list

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

- `__doc__`
- `__module__` and `__name__`
- `__dict__`

The docstring problem

Our beautiful replacement function lost

- 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)
```

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!

Example: configurable deprecated

Modify `deprecated` to take a message to print.

```
>>> @deprecated('function {f.__name__} is deprecated')
... def eot():
...     return 'EOT'
>>> eot()
function eot is deprecated
'EOT'
>>> eot()
'EOT'
```

Example: configurable deprecated

implementation as a class

```
class deprecated(object):
    def __init__(self, message):
        self.message = message

    def __call__(self, func):
        self.func = func
        self.count = 0
        return functools.update_wrapper(self.wrapper, func)

    def wrapper(self, *args, **kwargs):
        self.count += 1
        if self.count == 1:
            print self.message.format(f=self.func)
        return self.func(*args, **kwargs)
```

Example: configurable deprecated

implementation as a function

```
def deprecated(message):
    """print the message once and call the original function
      """
    def _deprecated(func):
        func.count = 0
        def wrapper(*args, **kwargs):
            func.count += 1
            if func.count == 1:
                print message.format(f=func)
            return func(*args, **kwargs)
        return functools.update_wrapper(wrapper, func)
    return _deprecated
```

Decorators work for classes too

- same principle

Decorators work for classes too

- same principle
- much less exciting
  - PEP 318 ⇒ “about 834,000 results”
  - PEP 3129 ⇒ “about 74,900 results”

Decorators work for classes too

- same principle
- much less exciting
  - PEP 318 ⇒ “about 834,000 results”
  - PEP 3129 ⇒ “about 74,900 results”

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

Example: plugin registration system

```
class WordProcessor(object):
    def process(self, text):
        for plugin in self.PLUGINS:
            text = plugin().cleanup(text)
        return text

    PLUGINS = []
    ...

@WordProcessor.plugin
class CleanMdashesExtension(object):
    def cleanup(self, text):
        return text.replace('&mdash;', u'\N{em dash}')
```

Decorators for methods

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

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

Decorators for methods

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

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

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

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

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
```

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

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

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

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
```

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`

The `property` triple: setter, getter, deleter

- attribute access `a.edge` calls `area.getx`
  - set with `@property`

The `property` triple: setter, getter, deleter

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

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`

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`

The property triple: setter, getter, deleter

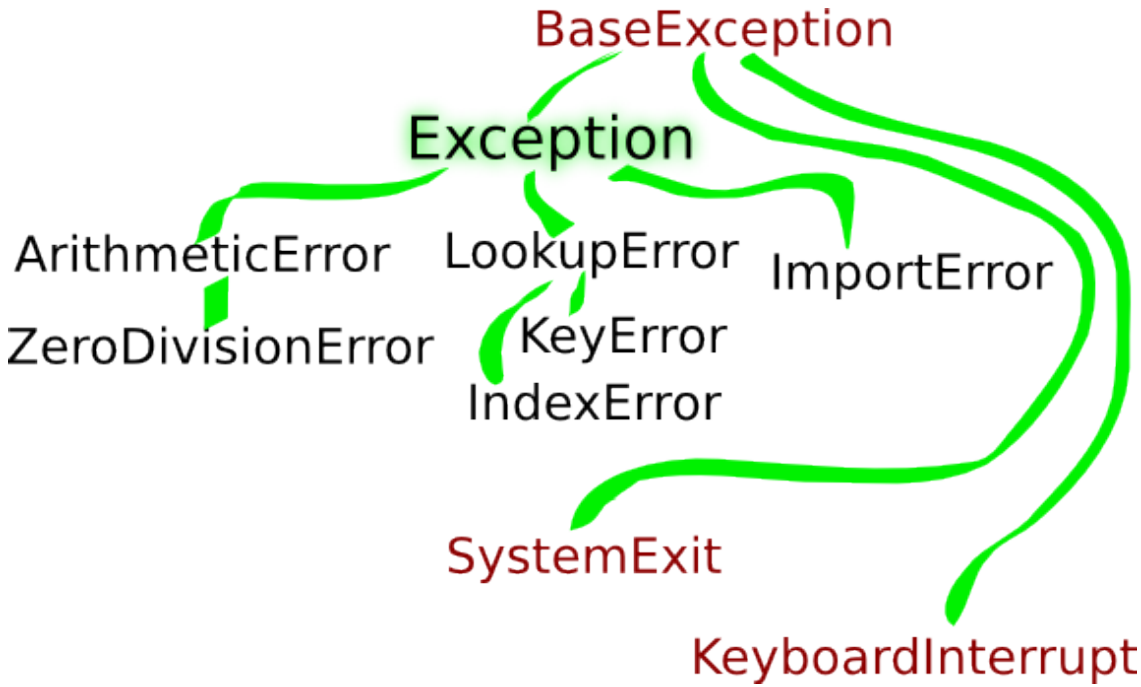
- attribute access `a.edge` calls `area.getx`
  - set with `@property`
- attribute setting `a.edge=3` calls `area.setx`
  - set with `.setter`
- attribute setting `del a.edge` calls `area.delx`
  - set with `.deleter`

Exception handling

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

Exception handling

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



Philosophical interludium

"timing is everything"

```
COMMITS = [132, 432, 050, 379]
def rm_change(change):
    if change in COMMITS:
        COMMITS.remove(change)


def rm_change(change):
    try:
        COMMITS.remove(change)
    except ValueError:
        pass
```

Philosophical interludium

"timing is everything"

```
COMMITS = [132, 432, 050, 379]
def rm_change(change):
    if change in COMMITS:
        COMMITS.remove(change)
```

L
B
Y
L

```
def rm_change(change):
    try:
        COMMITS.remove(change)
    except ValueError:
        pass
```

E
A
F
P

```
COMMITS = range(10**7)
rm_change(10**7); rm_change(10**7-1); rm_change(10**7-2)
```

Philosophical interludium

"timing is everything"

```
COMMITS = [132, 432, 050, 379]
def rm_change(change):
    if change in COMMITS:
        COMMITS.remove(change)
```

L O O K
B E F O R E
Y O U
L E A P

```
def rm_change(change):
    try:
        COMMITS.remove(change)
    except ValueError:
        pass
```

E A S I E R T O
A S K F O R
F O R G I V E N E S S T H A N
P E R M I S S I O N

```
COMMITS = range(10**7)
rm_change(10**7); rm_change(10**7-1); rm_change(10**7-2)
```

Freeing stuff in `finally`

How to make sure resources are freed?

```
resource = acquire()
try:
    do_something(resource)
finally:
    free(resource)
```

Acquiring resources

```
camera = CameraConnection('/dev/video1')
try:
    camera.powerup()
    picture = camera.take_picture()
except BaseException:
    camera.powerdown()
    raise
camera.powerdown()
```

Acquiring resources

```
camera = CameraConnection('/dev/video1')
try:
    camera.powerup()
    picture = camera.take_picture()
finally:
    camera.powerdown()
```

Nested finallys

```
try:
    try:
        print 'work'
        {}['???']
    finally:
        print 'finalizer a'
        1 / 0
finally:
    print 'finalizer b'
```

Nested finallys

```
try:
    try:
        print 'work'
        {}['???']
    finally:
        print 'finalizer a'
        1 / 0
finally:
    print 'finalizer b'
```

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

For completeness: else

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

```
try:
    ans = math.sqrt(num)
except ValueError:
    ans = float('nan')
else:
    print 'operation succeeded!'
```

Why are exceptions good?

```
# strip_comments.py
import sys
inp = open(sys.argv[1])
for line in inp:
    if not line.lstrip().startswith('#'):
        print line,
```

Why are exceptions good?

```
# strip_comments.py
import sys
inp = open(sys.argv[1])
for line in inp:
    if not line.lstrip().startswith('#'):
        print line,
```

A meaningful error message when:

- not enough arguments
- files cannot be opened

Using a context manager

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

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()
```

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.
```

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

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`
- `memoryview` (3.2)

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`
- `memoryview` (3.2)
- `decimal.localcontext`

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`
- `memoryview` (3.2)
- `decimal.localcontext`
- `warnings.catch_warnings`

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`
- `memoryview` (3.2)
- `decimal.localcontext`
- `warnings.catch_warnings`
- `contextlib.closing`

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`
- `memoryview` (3.2)
- `decimal.localcontext`
- `warnings.catch_warnings`
- `contextlib.closing`
- parallel programming
  - `concurrent.futures.ThreadPoolExecutor` (3.2)
  - `concurrent.futures.ProcessPoolExecutor` (3.2)

Context managers in the stdlib

- all file-like objects
  - `file`
  - `fileinput`, `tempfile` (3.2)
  - `bz2.BZ2File`, `gzip.GzipFile` `tarfile.TarFile`, `zipfile.ZipFile`
  - `ftplib`, `nntplib` (3.2 or 3.3)
- locks
  - `multiprocessing.RLock`
  - `multiprocessing.Semaphore`
- `memoryview` (3.2)
- `decimal.localcontext`
- `warnings.catch_warnings`
- `contextlib.closing`
- parallel programming
  - `concurrent.futures.ThreadPoolExecutor` (3.2)
  - `concurrent.futures.ProcessPoolExecutor` (3.2)
  - `nogil` (cython only)

Managing exceptions

```
class Manager(object):
    ...

    def __exit__(self, type, value, traceback):
        ...
        return swallow
```

Unittesting thrown exceptions

```
def test_indexing():
    try:
        {}['foo']
    except KeyError:
        pass
```

Unittesting thrown exceptions

```
def test_indexing():
    try:
        {}['foo']
    except KeyError:
        pass
```

Can we do better?

Unittesting thrown exceptions

```
def test_indexing():
    try:
        {}['foo']
    except KeyError:
        pass
```

Can we do better?

```
import py.test
def test_indexing():
    py.test.raises(KeyError, lambda:{}['foo'])
```

Unittesting thrown exceptions

```
def test_indexing():
    try:
        {}['foo']
    except KeyError:
        pass
```

Can we do better?

```
import py.test
def test_indexing():
    py.test.raises(KeyError, lambda:{}['foo'])
```

Can we do better?

Unittesting thrown exceptions

```
class assert_raises(object):
    def __init__(self, type):
        self.type = type
    def __enter__(self):
        pass
    def __exit__(self, type, value, traceback):
        if type is None:
            raise AssertionError('exception expected')
        if issubclass(type, self.type):
            return True
        raise AssertionError('wrong exception type')
```

Unittesting thrown exceptions

```
class assert_raises(object):
    def __init__(self, type):
        self.type = type
    def __enter__(self):
        pass
    def __exit__(self, type, value, traceback):
        if type is None:
            raise AssertionError('exception expected')
        if issubclass(type, self.type):
            return True
        raise AssertionError('wrong exception type')


def test_indexing():
    with assert_raises(KeyError):
        {}['foo']
```

Writing context managers as generators

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

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: `flushed`

```
@contextlib.contextmanager
def flushed(file):
    try:
        yield
    finally:
        file.flush()
```

assert_raises as a function

```
@contextlib.contextmanager
def assert_raises(exc):
    try:
        yield
    except exc:
        pass
    except Exception as value:
        raise AssertionError('wrong exception type')
    else:
        raise AssertionError(exc.__name__+' expected')
```

Summary

Summary

- Generators make iterators easy

Summary

- Generators make iterators easy
- Decorators make wrapping and altering functions and classes easy

Summary

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

That's all!