

Advanced Python

“Advanced Scientific Programming in Python”

Day 0

Trento — October 4th, 2010

Zbigniew Jędrzejewski-Szmek
zbyszek@in.waw.pl

Examples are in Python 3!

©2010, [CC BY-SA 3.0](#) ([source code available](#))

Press ‘t’ to toggle presentation/outline mode.

A plan

- decorators
- exception handling
- generators
- context managers
- py3k if `time.now() - 10:30 < 0`

After the lecture, some people complained that this was 'a dive in cold water' and that no introduction and not enough motivation was given. To make up for this deficiency, in the following paragraphs I'll try to convince you, the reader, that the presented techniques are actually useful, and that they help the programmer follow the Zen of Python.

The most important Python principle (import this) is DRY: **don't repeat yourself**. Decorators and context managers allow one to extract common parts of the code — function or class decoration and try/except/finally clauses — and to put them in a separate

Contents

- [A plan](#)
- [Decorators](#)
- [Decorators](#)
- [The decorator syntax](#)
- [A decorator doing smth. ...](#)
- [... written as a class](#)
- [Installing function wrappers](#)
- [Decorators can be stacked](#)
- [Example: deprecated2](#)
- [Example: deprecated2](#)
- [Raison d'être for decorators](#)
- [Mention of class decorators](#)
- [Example: plugin registration system](#)
- [Problems with decorators thus defined](#)
- [Exceptions](#)
- [Exception handling](#)
- [Philosophical interludium](#)
- [Going further than try/except](#)
- [Multiple finally levels](#)
- [For completeness: else](#)
- [Why are exceptions good?](#)

location.

Another part of the Zen of Python is **explicit is better than implicit**. The lengthy discussion on python-dev was caused by the desire to make the job of decoration simple, without repetitions, but visible when looking at the function header. Therefore decorator syntax puts the decorator in a line next to the function definition, so it's hard to miss, and on the outside because the decorator actually wraps the object, and thus this is the natural order which helps readability.

The Zen of Python also tells the programmer that **flat is better than nested** and **readability counts**. Decorators, generators, and context managers are all about splitting a complicated structure into two or more functions or classes, with each part performing a single job.

Of course one might argue that the rule **simple is better than complex** goes against the techniques described here, since they cannot really be said to be simple, maybe with the exception of generators. But once the decorator or context manager is defined, using it is *simple*. The definition is indeed **complex, but not complicated**, once the rules are understood. I would also argue, that those techniques do have inherent beauty, and **beautiful is better than ugly**.

- [Context Managers](#)
- [Context Manager Protocol](#)
- [Example](#)
- [Motto “batteries included”](#)
- [Motto “batteries included”](#)
- [Exceptions and context managers](#)
- [Generators](#)
- [Generators](#)
- [Communication is bi-directional!](#)
- [Communication](#)
- [Communication](#)
- [Aggravated communication](#)
- [Synthesis](#)
- [The return of the decorator](#)
- [Example: flushed](#)
- [Example: testing exceptions](#)
- [Example: testing exceptions, ctd.](#)
- [Python 3](#)
- [Python 3](#)
- [Conversion](#)

Decorators



Summary

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

Bruce Eckel

photo:
freefoto.com [CC](#)
[ND-NC-SA 3.0](#)

Decorators

- 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

The decorator *syntax*

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

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

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

A decorator doing smth. ...

... to remember the author and protect the V.I.I.P.

```
>>> @author('J.R.P.')
... def func(): pass
>>> func.author
'J.R.P.'
```

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

... written as a class

set an attribute on the function

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

The same written as nested functions:

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

```

return orig_f
return helper

```

Installing function wrappers

```

class deprecated:
    "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)

```

```

def deprecated__f(func):
    """Print a deprecation warning once

    Installs a wrapper function which
    prints a warning on first function use.

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

    """
    count = 0
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        if count == 1:
            print(func.__name__, 'is deprecated')
        return func(*args, **kwargs)
    return wrapper

```

Decorators can be stacked

```

@author('J.P.R.')
@deprecated()
def func(): pass

# old style
def func(): pass
func = author('J.P.R.')(
    deprecated(func))

```

Example: deprecated2

Modify deprecated to take a message to print.

```

>>> @deprecated2('function {func.__name__}'
...             ' is deprecated')
... def eot(): return 'EOT'
>>> eot()
function eot is deprecated
'EOT'
>>> eot()
'EOT'

```

Example: deprecated2

```

class deprecated2:
    def __init__(self, message):
        self.message = message # ←
    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:
            fmt = self.message.format # ←
            print(fmt(func=self.func)) # ←
        return self.func(*args, **kwargs)

```

The same written as a function.

```

def deprecated2(message):
    """
    >>> @deprecated2('function {func.__name__} is deprecated')
    ... def eot(): return 'EOT'
    """

```

```

>>> eot()
function eot is deprecated
'EOT'

''''
def _deprecated(func):
    count = 0
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        if count == 1:
            print(message.format(func=func))
        return func(*args, **kwargs)
    return wrapper
return _deprecated

```

Raison d'être for decorators

```

class K:
    def do_work(self, *args):
        pass

    @classmethod
    def create(cls, *args):
        return cls(*args)

    @staticmethod
    def method(*args):
        return 33

```

Mention of class decorators

same principle

much less exciting

- [PEP 318](#) \rightsquigarrow “about 834,000 results”
- [PEP 3129](#) \rightsquigarrow “about 74,900 results”

Example: plugin registration system

```

class WordProcessor:

```

```

PLUGINS = []
def process(self, text):
    for plug in self.PLUGINS:
        text = plug().cleanup(text)
    return text

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

```

Problems with decorators thus defined

tracebacks

introspection

repetition

Solutions

- `functools.update_wrapper(wrapper, wrapped)`
"Update a wrapper function to look like the wrapped function"
- module decorator
- [PEP 318](#) (function and method decorator syntax)
- [PEP 3129](#) (class decorator syntax)
- <http://wiki.python.org/moin/PythonDecoratorLibrary>
- <http://docs.python.org/dev/library/functools.html>
- <http://pypi.python.org/pypi/decorator>
- Bruce Eckel
 - [Decorators I](#): Introduction to Python Decorators
 - [Python Decorators II](#): Decorator Arguments
 - [Python Decorators III](#): A Decorator-Based Build System

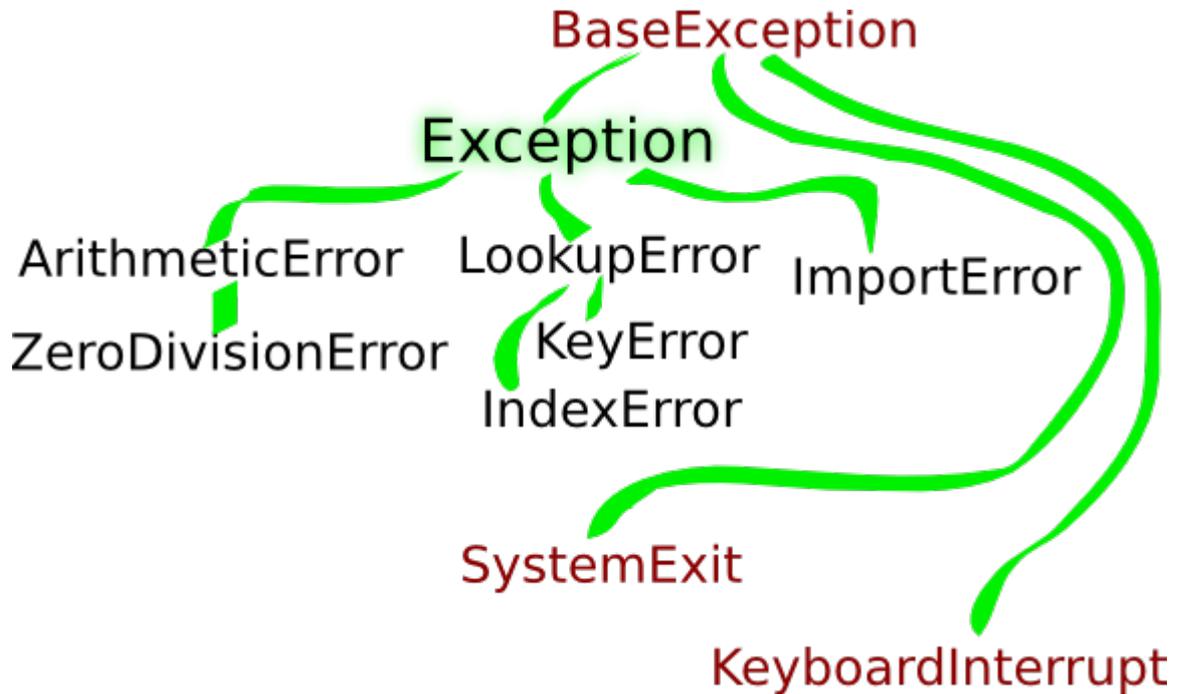
Exceptions



photo: flickr.com/photos/nickwheeleroz [CC BY-NC-SA 2.0](https://creativecommons.org/licenses/by-nc-sa/2.0/)

Exception handling

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



Philosophical interludium

```

CHANGES = list()
def remove_change_a(change):
    if change in CHANGES:
        CHANGES.remove(change)
    
```

L
B
Y
L

```

def remove_change_b(change):
    try: CHANGES.remove(change)
    except ValueError: pass
    
```

E
A
F
P

```

CHANGES = list(range(10**7))
r(10**7); r(10**7-1); r(10**7-2)
    
```

r = remove_change_a() ~~~~~> 1.46s
r = remove_change_b() ~~~~~> 0.85s

Going further than try/except

how to make sure resources are freed?

```

def add_user(login, uid, gid, name, home,
            shell='/bin/bash'):
    fields = (login, str(uid), str(gid),
    
```

```

    name, home, shell)
f = open(PASSWD_FILE, 'a')
fcntl.lockf(f.fileno(), fcntl.LOCK_EX)
try:
    f.seek(0, io.SEEK_END)
    f.write('.'.join(fields) + '\n')
    f.flush()
finally:
    fcntl.lockf(f.fileno(), fcntl.LOCK_UN)

```

Multiple finally levels

```

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

```

Nesting of finally levels is required if we want to make sure that all finalization statements are executed. When more than one statement is written in the same clause, an exception thrown by one of the statements would prevent the following ones from running. Putting each part in a separate level makes sure that all parts are run independently.

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])

```

```
out = open(sys.argv[2], 'w')
for line in inp:
    if not line.lstrip().startswith('#'):
        print(line, file=out)
```

A meaningful error message when:

- not enough arguments
- files cannot be opened

Context Managers



Context Manager Protocol

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

```
var = manager.__enter__()
try:
    do_something(var)
finally:
```

```
manager.__exit__()
```

Example

make sure output is written to file from buffers

```
with flushed(open('/etc/fstab', 'a')) as f:
    print('/dev/cdrom /cdrom', file=f)
os.system('mount /cdrom')
```

```
class flushed:
    def __init__(self, obj):
        self.obj = obj
    def __enter__(self):
        return self.obj
    def __exit__(self, *args):
        self.obj.flush()
```

Motto “batteries included”

```
with open(filename) as f:
    f.write(...)
```

- all file-like objects:
 - file → automatically closed
 - fileinput, tempfile (py>=3.2)
 - bz2.BZFile, gzip.GZFile, tar.TarFile, zip.ZipFile
 - ftplib → close connection (py>=3.2)

Motto “batteries included”

- locks
 - multiprocessing.RLock → automatically unlock
 - multiprocessing.Semaphore
 - memoryview → automatically released (py>=3.2)
- decimal.getlocalcontext → precision
- winreg.HKEY → open and close key
- warnings.catch_warnings → kill warnings
- contextlib.closes

Exceptions and context managers

An exception thrown in the with-block is given to `__exit__`. If `__exit__` returns

True, the exception is **swallowed**. Otherwise, the exception is **rethrown** after exiting all context managers.

Taken from `py.test` and `unittest.TestCase`.

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

class assert_raises:
    def __init__(self, type):
        self.type = type
    def __enter__(self):
        pass
    def __exit__(self, type, value, traceback):
        if type is None:
            raise AssertionError('no exc.')
        if isinstance(type, self.type):
            return True
        raise AssertionError('wrong exc.') from value
```

Generators



photo: freefoto.com [CC ND-NC-SA 3.0](https://creativecommons.org/licenses/by-nc-sa/3.0/)

Generators

```
>>> def words():  
...     print('generator running...')  
...     yield 'gugu'  
...     yield 'bebe'  
...     return  
>>> source = words()  
>>> for word in source:  
...     print('\N{rightwards arrow}', word)  
generator running...  
→ gugu  
→ bebe
```

- [Idiomatic Python by David Goodger](#)

Communication is bi-directional!

- generator can yield and raise
- the caller can throw, send, and close

images/caller-generator.svg

Communication

images/caller-generator-1.svg

```
>>> def generator():
...     print('executing ...', end=' ')
...     yield 'X'
...     Z = yield 'Y'
...     print('got', ans, '...', end=' ')
>>> g = generator()
>>> ans = next(g); print(ans)
executing ... X
>>> ans = next(g); print(ans)
Y
```

Communication

images/caller-generator-2.svg

```
>>> def generator():
...     print('executing ...', end=' ')
...     Z = yield; print('got', Z, '...', end=' ')
...     Z2 = yield; print('got', Z2)
>>> g = generator()
>>> ans = next(g); print(ans)
executing ... None
>>> ans = g.send('Z'); print(ans)
got Z ... None
```

Aggravated communication

```
>>> def strange():
...     try:
...         yield 'good'
...     except Exception as e:
...         yield 'bad'
```

```

...     yield 'done'
>>> gen = strange()
>>> next(gen)
'good'
>>> gen.throw(ValueError)
'bad'
>>> next(gen)
'done'

```

Synthesis

images/power_station.jpg

photo: freefoto.com [CC ND-NC-SA 3.0](https://creativecommons.org/licenses/by-nc-sa/3.0/)

The return of the decorator

```

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

```

Example: flushed

```

class flushed:
    def __init__(self, obj):
        self.obj = obj
    def __enter__(self):
        return self.obj
    def __exit__(self, *args):
        self.obj.flush()

```

```

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

```

Example: testing exceptions

```

class assert_raises:
    def __init__(self, type):
        self.type = type
    def __enter__(self):
        pass
    def __exit__(self, type, value, traceback):
        if type is None:
            raise AssertionError('no exc. raised')
        if isinstance(type, self.type):
            return True
        raise AssertionError(
            'wrong exc. raised') from value

```

Example: testing exceptions, ctd.

```

@contextlib.contextmanager
def assert_raises_2(type):
    try:
        yield
    except type:
        return
    except Exception as value:
        raise AssertionError(
            'wrong exc. raised') from value
    else:
        raise AssertionError(
            'exception was not raised')

```

```

>>> class assert_raises:
...     def __init__(self, type):
...         self.type = type
...     def __enter__(self):
...         pass
...     def __exit__(self, type, value, traceback):
...         if type is None:
...             raise AssertionError('exception was not raised')
...         if isinstance(type, self.type):
...             return True
...         raise AssertionError('wrong exc. was raised') from value...
>>>
with assert_raises(KeyError): {}['key']
...
>>> with assert_raises(KeyError): 1/0
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

```

ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 11, in __exit__

AssertionError: wrong exc. was raised

```
>>> with assert_raises(KeyError): pass
```

...

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 8, in __exit__

AssertionError: exception was not raised

```
>>> @contextlib.contextmanager
```

```
... def assert_raises(type):
```

```
...     try:
```

```
...         yield
```

```
...     except type:
```

```
...         return
```

```
...     except Exception as value:
```

```
...         raise AssertionError('wrong exc. was raised') from value...
```

```
    else:
```

```
...         raise AssertionError('exception was not raised')
```

...

```
>>> with assert_raises(KeyError): {'key'}
```

...

```
>>> with assert_raises(KeyError): 1/0
```

...

Traceback (most recent call last):

File "<stdin>", line 4, in assert_raises

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/usr/local/lib/python3.2/contextlib.py", line 46, in __exit__

self.gen.throw(type, value, traceback)

File "<stdin>", line 8, in assert_raises

AssertionError: wrong exc. was raised

```
>>> with assert_raises(KeyError): pass
```

...

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

```
File "/usr/local/lib/python3.2/contextlib.py", line 35, in __exit__
    next(self.gen)
File "<stdin>", line 10, in assert_raises
AssertionError: exception was not raised
```

Python 3

images/python.jpg

photo: flickr.com/photos/nasmac [CC BY-SA 2.0](#)

Python 3

currently 3.2

- str/bytes ↔ unicode/str
 - new string formatting
- keywords
 - nonlocal
 - as, from
 - with
- module changes

Conversion

“good Python 2.6 code is not very different from Python 3.0”

2to3

```
~/mdp$ python3 setup.py install
```

```
Converting to Python3 via 2to3...
```

```
RefactoringTool: Skipping implicit fixer: buffer
```

```
...
```

```
RefactoringTool: Refactored build/py3k/mdp/test/test_parallelhinet.py
```

```
RefactoringTool: Refactored build/py3k/mdp/parallel/parallelhinet.py
```

```
RefactoringTool: Files that were modified:
```

```
RefactoringTool: build/py3k/mdp/test/test_parallelhinet.py
```

```
RefactoringTool: build/py3k/mdp/parallel/parallelhinet.py
```

```
running install
```

```
...
```

```
Writing /usr/lib/python3.1/site-packages/MDP-2.6-py3.1.egg-info
```

3to2