Advanced features

Amit Kotlovski
Open-Source Consultancy
http://amitkot.com

Agenda

- Configurable decorators
- Context managers

Decorator functions - Reminder

- Functions that:
  - get a function as an argument
  - return a new function wrapping the original one
  - (usually) add some functionality

Decorators have an additional special syntax in Python

```python
from functools import wraps

def logme(fn):
    @wraps(fn)
    def internal(*args, **kwargs):
       print('--- {} started’.format(fn.__name__))
        res = fn(*args, **kwargs)
        print('--- {} finished’.format(fn.__name__))
        return res
    return internal

@logme
def print_word(word="word"):
    print(word)
```

```
>>> print_word('kitchen')
--- print_word started
kitchen
--- print_word finished
```

Passing arguments to decorators

- Decorator functions can be passed arguments
    - requires another layer of wrapping (next example)

Decorator with argument

```python
from functools import wraps

def logme_level(level='INFO'):
    def wrapper(fn):
        @wraps(fn)
        def internal(*args, **kwargs):
           print('--- {} {} started'.format(level, fn.__name__))
            res = fn(*args, **kwargs)
            print('--- {} {} finished'.format(level, fn.__name__))
            return res
        return internal
    return wrapper
```

```
@logme_level()
def print_1_word(word="word"):
    print(word)

@logme_level(level="DEBUG")
def print_2_words(first="first", second="second"):
    print(first, second)
```

```
>>> print_1_word(‘nice')
--- INFO print_1_word started
nice
--- INFO print_1_word finished

>>> print_1_word()
--- INFO print_1_word started
word
--- INFO print_1_word finished

>>> print_2_words('casual', ‘fruit')
--- DEBUG print_2_words started
casual fruit
--- DEBUG print_2_words finished
```

@decorator()

```python
@logme_level()
def print_1_word(word="word"):
    print(word)
```

Bonus: Optional argument

- We can implement a decorator with an optional argument

- Decorator's () not required when using argument's default value

Bonus: Optional argument

- If no argument is passed - it behaves like the previous example
- Otherwise it adds another wrapper layer

```python
from functools import wraps

def logme_level(fn=None, level='INFO'):
    def actual_wrapper(fn):
        @wraps(fn)
        def internal(*args, **kwargs):
            print('--- {} {} started'.format(level, fn.__name__))
            res = fn(*args, **kwargs)
            print('--- {} {} finished'.format(level, fn.__name__))
            return res
        return internal
    if fn is None:  # called with explicit log level argument
        def waiting_for_fn(fn):
            return actual_wrapper(fn)
        return waiting_for_fn
    else:
        return actual_wrapper(fn)
```

```
@logme_level
def print_1_word(word="word"):
    print(word)

@logme_level(level="DEBUG")
def print_2_words(first="first", second="second"):
    print(first, second)

>>> print_1_word(‘nice')
--- INFO print_1_word started
nice
--- INFO print_1_word finished

>>> print_2_words('casual', ‘fruit')
--- DEBUG print_2_words started
casual fruit
--- DEBUG print_2_words finished
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Context managers

Opening a file

```
f = open('/etc/passwd')    # (1) initialise
try:
    use_file_object(f)     # (2) do things
finally:
    f.close()              # (3) clean-up
```

Opening a file

```
f = open('/etc/passwd')      # (1) initialise
try:
    use_file_object(f)       # (2) do things
finally:
    f.close()                # (3) clean-up
```

```
Markered - Interesting parts
Not markered - Boilerplate
```

Context manager

- Hides the boilerplate of
  - the initialisation
  - the clean up
- PEP 343 -- The “with” statement

With a context manager

```
var = manager.__enter__()
try:
    do_things(var)
finally:
    manager.__exit__()
```

Using a context manager

```python
with open('/etc/passwd') as f:
    do_things(f)
```

Context manager protocol

- Define `__enter__()` and `__exit__()` methods

Defining a context manager

```
class always_close(object):
    def __init__(self, thing):
        print('__init__')
        self.thing = thing
    def __enter__(self):
        print('__enter__')
        return self.thing
    def __exit__(self, *args):
        print('__exit__')
        self.thing.close()

>>> with always_close(open('/etc/passwd')) as f:
...     print(len(f.readlines()))
__init__
__enter__
86
__exit__
```

`contextlib` module

- Provides utilities for building context managers
- @contextmanager decorator facilitates writing context managers using generator functions

Using contextlib

```
from contextlib import contextmanager

@contextmanager
def always_close(thing):
    try:
        print('--- __enter__')
        yield thing
    finally:
        print('--- __exit__')
        thing.close()

>>> with always_close(open('/etc/passwd')) as f:
...     print(len(f.readlines()))
--- __enter__
86
--- __exit__
```

Some examples of context managers

Multithread lock

Elegantly maintaining a lock in critical sections

```python
import threading

lock = threading.Lock()

with lock:
    print("executing code while holding a lock")
```

Unit tests

Asserting that code under test raises specific Exceptions

```
import unittest

class JoinTest(unittest.TestCase):
    def test_join_ints(self):
        with self.assertRaises(TypeError):
            ', '.join([1, 2, 3])

    def test_join_ints_check_message(self):
        with self.assertRaisesRegex(TypeError,
                                    '.*int found$'):
            ', '.join([1, 2, 3])
```

Database connections and transactions

On errors the whole transaction can be rolled back

```
import sqlite3

with sqlite3.connect(':memory:') as conn:
    # start a new db transaction
    conn.execute(
        'create table students (id int primary key,'
                               'name char(50))')
    conn.execute('insert into students values (?,?)',
                 (0, 'Urusula LeGuin'))
```

Django Database transactions

Django code can perform atomic database transactions using the `@transaction.atomic` context manager

```
from django.db import transaction

def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

Temporary files

```python
import tempfile

with tempfile.NamedTemporaryFile() as f:
    print('Writing to tempfile:', f.name)
    f.write(b'Some data')
    f.flush()
```

```
# Output:
Writing to tempfile: /tmp/tmp7ly_7sd2
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Summary

- We've seen some more decorator configuration
- Context managers help minimise boilerplate code

QUESTION
EVERYTHING

Thanks!

twitter	@amitkot
www	amitkot.com