Lecture 5: Object-Oriented Programming

Chirag Sakhuja

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

"The Zen of Python" – Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

List comprehension functionality

```
[1, 2, 3, 4, 5]
[f(x) for x in iterable]
[f(1), f(2), f(3), f(4), f(5)]
```

Lambda syntax

Arguments

```
lambda x, y: x * y
```

Operation

LBE: Sum of first N even squares

```
from functools import reduce

sq = map(lambda x: x ** 2, range(N + 1))
even_sq = filter(lambda x: x % 2 == 0, sq)
sum = reduce(lambda x, y: x + y, even_sq)
# => 20 if N = 5
```

Notice we didn't convert to lists in between operations
Map and Filter objects are iterable

LBE: Fibonacci number generator

```
def fib_gen():
    a, b = 0, 1
    while True:
        a, b = b, a + b
        yield a
gen = fib_gen(3)
for fib in gen:
    if fib > N: break
    print(fib, end=',')  # => 1,1,2,3,5, if N = 5
```

Infinitely generate Fibonacci numbers

Lazy evaluation!

String filter

```
def filter_co(pattern):
    print('Searching for {}'.format(pattern))
    while True:
        line = yield
        if pattern in line:
            print(line)

co = filter_co('help')
next(co)                          # => Searching for help
co.send('Please send help')       # => Please send help
co.send('This is easy')           # =>
co.send("You don't need help!")   # => You don't need help!
```

Function factory

```python
def factory(x):
    def helper(y):
        return x + y
    return helper

add1 = factory(1)
add2 = factory(2)
add1(10)         # => 11
add2(10)         # => 12
factory(3)(10)   # => 13
```

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

Decorators are fancy wrappers

- Decorators wrap functions
- Defined as a function factory
- Invoked using the @ operator above the function you want to decorate

Basic function

```
def foo(x, y):
    print(x + y)

foo(1, 2)    # => 3
```

Debug decorator

```
def debug(func):
    def wrapper(*args, **kwargs):
        print('Arguments: ', args, kwargs, end=' -> ')
        return func(*args, *kwargs)
    return wrapper

def foo(x, y):
    print(x + y)

foo(1, 2)               # => 3
foo_debug = debug(foo)
foo_debug(1, 2)         # => Arguments: (1, 2) {} -> 3
```

Do some stuff, then call the function and return the result (forwarding in the arguments)

This looks ugly...

LBE: Debug decorator

```python
def debug(func):
    def wrapper(*args, **kwargs):
        print('Arguments: ', args, kwargs, end=' -> ')
        return func(*args, *kwargs)
    return wrapper

@debug
def foo(x, y):
    print(x + y)

foo(1, 2)     # => Arguments: (1, 2) {} -> 3
```

LBE: Coroutine priming decorator

```
def coroutine(func):
    def wrapper(*args, **kwargs):
        co = func(*args, **args)
        next(co)
        return co
    return wrapper
@coroutine
def print_co():
    while True:
        val = yield
        print('Received: {}'.format(val))
co = print_co()
co.send(1)    # => Received: 1
```

Initialize the coroutine, prime it with the next operator, then return the primed coroutine

No need to call next when using the coroutine now!

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

Classes

There is a difference!

- Classes encapsulate state and functions/methods
- Classes in Python are syntactic sugar built upon dictionaries under the hood
  - Relevant in some cases, "implementation details" in most cases
- No notion of public/private/protected
  - Everything is necessarily public!

Class definitions

- Class object: a definition that encapsulates attributes and functions
- Class instance: a "copy" of a class object that has its own state
  - Created by calling a constructor defined in a class object
- Class attributes: variables encapsulated by a class object that are the same for all class instances and can be directly accessed via the class object (similar to `static` in C++)
- Instance attributes: variables encapsulated by a class object that have independent values for each class instance

Class objects

- Think of a class object as a template for how to instantiate a class instance
- In C++, a "class object" is a "class definition"

Class object syntax

Define a class

```python
class MyClass:
    # ...
```

Body of the class

Class attributes in a class object

```
class MyClass:
    class_attribute = 10

type(MyClass)                     # => <class 'type'>
print(MyClass.class_attribute)    # => 10
MyClass.class_attribute = 8
print(MyClass.class_attribute)    # => 8
```

Only one class_attribute exists

Accessed through the class object name using the dot operator

Functions in a class object

```
class MyClass:
    class_attribute = 10

    def func():
        print('Hello from MyClass')

MyClass.func()    # => Hello from MyClass
```

Accessed through the class object name using the dot operator

Functions vs methods

- Functions can be defined in any scope
  - In global scope, as we've seen in the past
  - Inside other functions, as we've seen in the paste
  - Inside class objects
- "Functions" that have a special parameter, `self`, through which they can access a class instance's state is called a method
  - Methods are what you commonly think of when using an object-oriented paradigm

Methods

Called a method because it references an instance

```
class MyClass:
    def func(self):
        print('Hello from MyClass')

type(MyClass)        # => <class 'type'>
inst = MyClass()
type(inst)           # => <class '__main__.MyClass'>
inst.func()          # => Hello from MyClass
MyClass.func(inst)   # => Hello from MyClass
```

Create class instance from class object (i.e. instantiate)

Nothing special about methods

- Methods are just a term for a function that follows a specific rule
  - The rule being that the first argument is `self`
  - `self` refers to the instance of the class object

Where `class_inst` is an instantiation of `ClassObject`

```
class_inst.func(args)
```

turns into

```
ClassObject.func(class_inst, args)
```

Constructors and instance attributes

- Constructors are where you define instance attributes
- The constructor in Python has the name `__init__`
- The constructor is a method
- Multiple constructors with different arguments can be used

Instance attributes

Define a constructor

```
class MyClass:
    def __init__(self):
        self.inst_attr = 10
```

Somewhat strange, but you simultaneously declare and define instance attributes in the constructor

```
inst = MyClass()
print(inst.inst_attr)       # => 10
print(MyClass.inst_attr)    # => AttributeError!
```

Instance attributes are specific to the class instance

LBE: Car information

```python
class Car:
    total_cars = 0
    def __init__(self, make='Toyota', model='Camry'):
        Car.total_cars += 1
        self.make = make
        self.model = model
car = Car()
dream_car = Car('McLaren', 'P1')
print(car.make)           # => Toyota
print(dream_car.make)     # => McLaren
print(Car.total_cars)     # => 2
```

LBE: Complex numbers

Need to import math

```
class Complex:
    def __init__(self, real=0, imag=0):
        self.real = real
        self.imag = imag
    def getRho(self):
        return math.hypot(self.real, self.imag)
    def getTheta(self):
        return math.atan2(self.imag, self.real)
c = Complex(1, 1)
print(c.getRho(), c.getTheta())  # => 1.414… 0.785…
```

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

Inheritance

- Imagine stacking one class instance's scope onto another class instance
  - Since data is stored in dictionaries under the hood, what's really happening is just that the dictionaries for each scope are being appended
  - The "nearest" scope is the one that gets used (i.e. the child class's scope)
- No need to use any explicit keyword, such as `virtual`, to get polymorphic behavior
- Parent constructors are not implicitly called

```
class Child(Base):
    # …
```

Polymorphism

```
class Vehicle:
    def __init__(self):
        print('Constructing Vehicle')
    def func(self):
        print('func from Vehicle')
class Car(Vehicle):
    def __init__(self):
        print('Constructing Car')
    def func(self):
        print('func from Car')
c = Car()     # => Constructing Car
c.func()      # => func from Car
```

Comma-delimited class objects from which to inherit

Instance attributes are not implicitly inherited

```
class Vehicle:
    def __init__(self, speed=0):
        self.speed = speed
        print('Constructing Vehicle')
    def func(self):
        print('func from Vehicle')
class Airplane(Vehicle):
    def __init__(self):
        print('Constructing Airplane', self.speed)
a = Airplane()     # => AttributeError!
a.func()
```

Not implicitly created

Instance attributes are not implicitly inherited

```python
class Vehicle:
    def __init__(self, speed=0):
        self.speed = speed
        print('Constructing Vehicle')
    def func(self):
        print('func from Vehicle')
class Airplane(Vehicle):
    def __init__(self):
        super(Airplane, self).__init__(10)
        print('Constructing Airplane', self.speed)
a = Airplane()    # => Constructing Vehicle\nConstructing Airplane 10
a.func()          # => func from Vehicle
```

Multiple inheritance

- Term used when a class inherits from more than one base class
- Scopes are built up going left-to-right in the inheritance list

```
class Child(Base1, Base2, ..., BaseN):
    # …
```

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

Magic methods

- `__init__` is a special Python method; are there others?
- `__add__`
- `__iter__`
- `__next__`
- `__len__`
- `__lt__`
- `__str__`
- …

LBE: Pretty printing

```python
class Complex:
    def __init__(self, real=0, imag=0):
        self.real = real
        self.imag = imag


c = Complex(1, 2)
print(c) # => <__main__.Complex object at 0x7f1785f040b8>
```

LBE: Pretty printing

```python
class Complex:
    def __init__(self, real=0, imag=0):
        self.real = real
        self.imag = imag
    def __str__(self):
        return '({}, {})'.format(self.real, self.imag)

c = Complex(1, 2)
print(c) # => (1, 2)
```

LBE: Adding custom objects

```python
class Complex:
    def __init__(self, real=0, imag=0):
        self.real = real
        self.imag = imag
    # ...
    def __add__(self, other):
        return Complex(self.real + other.real,
                       self.imag + other.imag)
c1 = Complex(1, 2)
c2 = Complex(2, 3)
print(c1 + c2)    # => (3, 5)
```

Building a **range** class

```python
for i in range(3):
    print(i, sep=', ')    # => 0, 1, 2
```

Building a range class

```python
def range(n):
    i = 0
    while i < N:
        yield i
        i += 1

for i in range(3):
    print(i, sep=', ')    # => 0, 1, 2
```

LBE: Building a range class

```
class range:
    def __init__(self, n):
        self.n = n
        self.i = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            tmp = self.i
            self.i += 1
            return tmp
        else:
            raise StopIteration
```

```
for i in range(3):
    print(i, sep=', ')    # => 0, 1, 2
```

Any generator can be written as a class

Generators are much more concise though!

We'll look at this shortly, but it's how you notify the caller that the iterator is expended

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

Exception handling

- Exceptions happen when there is some sort of error at runtime
- Exceptions are not-so-exceptional in Python
  - Exceptions are useful, so we want to acknowledge and handle them
  - "Ask for forgiveness, not permission"
- Use `try…except` block
- Exceptions are just objects

`try` statement

- Attempt executing the `try` block
- If there is an exception, immediately search for an `except` block that matches (including derived exceptions)
  - If there is a match, execute it
  - If there is no match, continue raising until the exception reaches the caller
- If there is no exception, jump to the code after the `try` block
  - May be an `else` block (distinct from the `else` in an `if…else` block)
  - May also be a `finally` block

We'll skip over these because they can be looked up and I personally haven't heard a convincing reason to use them

Basic error checking

```
float(input('Enter the first operand: '))
# => 'Enter the first operand: ' <= Hi
# => ValueError!
```

Basic error checking

```python
while True:
    try:
        float(input('Enter the first operand: '))
        break
    except ValueError:
        print('Try again!')
# => Enter the first operand: <= Hi
# => Try again!
# => Enter the first operand: <= 2
```

Lazy error checking

```
try:
    float(input('Enter the first operand: '))
except:
    print('Try again!')
```

Catches all errors

Don't do this unless you enjoy being a lazy programmer

Custom exceptions are good

```
class TooFewWheelsException(Exception):
    pass

class Car:
    def __init__(self, wheels=0):
        if wheels < 0:
            raise TooFewWheelsException()
        self.wheels = wheels

c = Car(-1)
```

Custom exceptions inherit from existing exceptions (where base class is Exception)

raise keyword is how we create an exception

Make helpful custom exceptions

```python
class TooFewWheelsException(Exception):
    def __str__(self):
        return 'Good luck driving like this!'

class Car:
    # ...

try:
    c = Car(-1)
except TooFewWheelsException as e:
    print(e)    # => Good luck driving like this!
```

Notice as keyword, which saves the exception in a variable so it can be used

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

Tenets of OOP

- Encapsulation: grouping together related data and operations into a single component
- Abstraction: modeling entities by exposing a well-formed and logical interface to the encapsulated fields
- Inheritance: creating an 'is a' relationship between base and derived classes
- Polymorphism: customizing the behavior of a derived class to take its own added fields into account

Good OOP is a lot of work

- Proper object-oriented programming requires following two principles
  - Cohesion: the focus of an individual component
  - Coupling: the relationship between individual components
- High cohesion means everything related to a component is encapsulated within that component, and everything else is encapsulated by other components
- Loose coupling means that components only depend on other components when necessary
- We want high cohesion and loose coupling so changing a component doesn't affect other components or require rethinking the abstraction

OOP conventions

- Class/instance attributes should be nouns
- Methods should be verbs
- In Python: prefix "private" attributes, functions, and methods with an underscore, since there is no language construct
  - This is only convention; nothing is enforcing it

Consider when to use OOP

- Previous slides just contain buzzwords
  - Buzzwords let you communicate with others
  - Buzzwords are not the be-all and end-all
- General rule of thumb: if it's a small project or only a few people are working on it, OOP may not be necessary
- Good OOP is hard, bad OOP gets in the way

Overview

Recap

Decorators

Classes

Inheritance

Magic methods

Exceptions

Good OOP practices

Conclusion

Key takeaways

- Python has full OOP support, even though it is somewhat different
- OOP is not always the best approach

References

[1] http://stanfordpython.com/

[2] https://www.jasoncoffin.com/cohesion-and-coupling-principles-of-orthogonal-object-oriented-programming/

[3] https://codingarchitect.wordpress.com/2006/09/27/four-tenets-of-oop/