Python for Finance

Advanced Features

Andras Niedermayer

DAUPHINE
UNIVERSITÉ PARIS

Objects of Interest

- object oriented programming (Wikipedia)

Definition (Object-oriented Programming)

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.

My First Class

- define an example class

```
class ExampleClass(object):
    """Just a simple example"""

    def __init__(self):
        """This is the constructor method"""
        self.my_integer = 1
        self.my_string = 'red'

    def print_stuff(self):
        print(self.my_integer)
        print(self.my_string)
```

- construct an object t which is an instance of class ExampleClass

```
t = ExampleClass()
```

- call method print_stuff of object t

```
t.print_stuff()
```

Help

- help!

```
help(ExampleClass)
```

- or help for specific object:

```
help(t)
```

- help for a method of an object:

```
help(t.print_stuff)
```

- in IPython you can also write

```
ExampleClass?
t?
t.print_stuff?
```

Special Methods

- special methods:
  - __repr__
  - __str__
  - __getattr__
  - __add__, __radd__, __iadd__
  - and many others

Represenation

- you can make an object printable with the __repr__ method

```
class UselessMatrix(object):
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return 'my name is {}'.format(
            self.name
        )
```

- print!

```
U1 = UselessMatrix('foo')
print(U1)
```

- __repr__ is implicitly called when you print an object, so the above could also have been written explicitly as

```
U1 = UselessMatrix('foo')
print(U1.__repr__())
```

Adding Stuff

- you can make an object addable with the __add__ method

```
class UselessMatrix(object):
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return 'my␣name␣is␣{}'.format(
            self.name)
    def __add__(self, other):
        return UselessMatrix(
          '({}␣plus␣{})'.format(
            self.name, other.name
        ))
```

Adding Stuff

- now we can add two "UselessMatrix"es with the __add__ method:

```
U1=UselessMatrix('foo')
U2=UselessMatrix('bar')
U1.__add__(U2)
```

- but it's tedious to write U1.__add__(U2). So for the special case of the __add__ method you can also use the "+" sign as a shortcut. So the above can also be written as:

```
U1=UselessMatrix('foo')
U2=UselessMatrix('bar')
U1+U2
```

Excercise

1. Create a Vector class with attributes “x” and “y”. Define a method for adding two vectors, a method to compute the length of a vector, and a method to represent a vector. Create vectors $v_1 = (2, 0)$ and $v_2 = (1, 4)$. What is the length of $v_1 + v_2$?

Solution

- Excercise 1

```
class Vector(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):
        """
        'v1+v2' calls this method with
        self==v1 and other==v2
        """
        return Vector(self.x+other.x,
                      self.y+other.y)
    def length(self):
        return (self.x**2 + self.y**2)**0.5
    def __repr__(self):
        return 'Vector(x={}, y={})'.format(
            self.x, self.y)

v1=Vector(2,0)
v2=Vector(1,4)
print(v1+v2)
print((v1+v2).length())
```

Inheritance

- inheritance lets us reuse classes and slightly modify their behavior
- a person knows to say hello

```
class Person(object):
    def __init__(self, name, first_name):
        self.name = name
        self.first_name = first_name
    def __str__(self):
        return self.first_name + " " \
            + self.name
    def hello(self):
        print('Hello, my name is ' \
            + str(self))
p=Person('Smith', 'John')
```

Inheritance

- but a special agent says hello differently!

```
class SpecialAgent(Person):
    def __init__(self, name,
                 first_name, number):
        Person.__init__(self, name,
                        first_name)
        self.number = number
    def __str__(self):
        return '{0}, {1} {0}.'\
               ' I am agent {2}.'.format(
               self.name, self.first_name,
               self.number)
p2=SpecialAgent('Bond', 'James', '007')
p2.hello()
```

- observe that "hello" is not defined in class SpecialAgent but can still be used because SpecialAgent is a subclass of Person

Excercise

1. Create a class Animal which has the method "speak_twice", which simply calls the method "speak" twice in a row. Create a class Dog that prints "woof" when it speaks and a class Cat which prints "meow" when it speaks. Create an animal farm with dogs and cats and let them speak twice:

```
animal_farm = [Dog(), Cat(), Dog()]
for animal in animal_farm:
    animal.speak_twice()
```

Solution

- Excercise 1

```
class Animal(object):
    def speak_twice(self):
        self.speak()
        self.speak()
class Dog(Animal):
    def speak(self):
        print('woof!')
class Cat(Animal):
    def speak(self):
        print('meow!')
animal_farm = [Dog(), Cat(), Dog()]
for animal in animal_farm:
    animal.speak_twice()
```

Functions as parameters

- define two functions

```
def add_one(x):
    return x+1

def multiply_by_two(x):
    return 2*x
```

- define a function that applies another function twice

```
def apply_twice(f, y):
    return f(f(y))
```

- call function that applies "add_one" and "multiply_by_two" twice

```
apply_twice(add_one, 10)
apply_twice(multiply_by_two, 10)
```

Anonymous Functions

- if you are too lazy to define a function "add_one", just create an anonymous function

```
apply_twice(lambda x: x+1, 10)
```

Comprehensions

- list comprehensions generate a list

```
[x+1 for x in [10, 20, 30]]
```

- list comprehensions with conditions

```
[x+1 for x in [10, 20, 30] if x!=20]
```

This could be read as the mathematical notation $\{x+1 | x \in \{10, 20, 30\}, x \neq 20\}$ except that we have a list and not a set

- but this also works for sets (set comprehensions):

```
{x+1 for x in [10, 20, 30] if x!=20}
```

- and dictionaries (dictionary comprehensions):

```
{x:x+1 for x in [10, 20, 30] if x!=20}
```

- and tuples (tuple comprehensions):

```
tuple(x+1 for x in [10, 20, 30] if x!=20)
```