Practical Python Programming

A course by @dabeaz

Contents | Previous (3.6 Design discussion) | Next (4.2 Inheritance)

4.1 Classes

This section introduces the class statement and the idea of creating new objects.

Object Oriented (OO) programming

A Programming technique where code is organized as a collection of *objects*.

An *object* consists of:

- Data. Attributes
- Behavior. Methods which are functions applied to the object.

You have already been using some OO during this course.

For example, manipulating a list.

```
>>> nums = [1, 2, 3]
>>> nums.append(4)      # Method
>>> nums.insert(1,10)   # Method
>>> nums
[1, 10, 2, 3, 4]        # Data
>>>
```

`nums` is an *instance* of a list.

Methods (`append()` and `insert()`) are attached to the instance (`nums`).

The `class` statement

Use the `class` statement to define a new object.

```
class Player:
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y
        self.health = 100

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def damage(self, pts):
        self.health -= pts
```

In a nutshell, a class is a set of functions that carry out various operations on so-called *instances*.

Instances

Instances are the actual *objects* that you manipulate in your program.

They are created by calling the class as a function.

```
>>> a = Player(2, 3)
>>> b = Player(10, 20)
>>>
```

`a` and `b` are instances of `Player`.

Emphasize: The class statement is just the definition (it does nothing by itself). Similar to a function definition.

Instance Data

Each instance has its own local data.

```
>>> a.x
2
>>> b.x
10
```

This data is initialized by the `__init__()`.

```
class Player:
    def __init__(self, x, y):
        # Any value stored on `self` is instance data
        self.x = x
```

```
        self.y = y
        self.health = 100
```

There are no restrictions on the total number or type of attributes stored.

Instance Methods

Instance methods are functions applied to instances of an object.

```
class Player:
    ...
    # `move` is a method
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

The object itself is always passed as first argument.

```
>>> a.move(1, 2)

# matches `a` to `self`
# matches `1` to `dx`
# matches `2` to `dy`
def move(self, dx, dy):
```

By convention, the instance is called `self`. However, the actual name used is unimportant. The object is always passed as the first argument. It is merely Python programming style to call this argument `self`.

Class Scoping

Classes do not define a scope of names.

```
class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def left(self, amt):
        move(-amt, 0)       # NO. Calls a global `move` function
        self.move(-amt, 0)  # YES. Calls method `move` from above.
```

If you want to operate on an instance, you always refer to it explicitly (e.g., `self`).

Exercises

Starting with this set of exercises, we start to make a series of changes to existing code from previous sections. It is critical that you have a working version of Exercise 3.18 to start. If you don't have that, please work from the solution code found in the `Solutions/3_18` directory. It's fine to copy it.

Exercise 4.1: Objects as Data Structures

In section 2 and 3, we worked with data represented as tuples and dictionaries. For example, a holding of stock could be represented as a tuple like this:

```
s = ('GOOG',100,490.10)
```

or as a dictionary like this:

```
s = { 'name'   : 'GOOG',
      'shares' : 100,
      'price'  : 490.10
}
```

You can even write functions for manipulating such data. For example:

```
def cost(s):
    return s['shares'] * s['price']
```

However, as your program gets large, you might want to create a better sense of organization. Thus, another approach for representing data would be to define a class. Create a file called `stock.py` and define a class `Stock` that represents a single holding of stock. Have the instances of `Stock` have `name`, `shares`, and `price` attributes. For example:

```
>>> import stock
>>> a = stock.Stock('GOOG',100,490.10)
>>> a.name
'GOOG'
>>> a.shares
100
>>> a.price
```

```
490.1
>>>
```

Create a few more `Stock` objects and manipulate them. For example:

```
>>> b = stock.Stock('AAPL', 50, 122.34)
>>> c = stock.Stock('IBM', 75, 91.75)
>>> b.shares * b.price
6117.0
>>> c.shares * c.price
6881.25
>>> stocks = [a, b, c]
>>> stocks
[<stock.Stock object at 0x37d0b0>, <stock.Stock object at 0x37d110>, <stock.Stock object at
>>> for s in stocks:
     print(f'{s.name:>10s} {s.shares:>10d} {s.price:>10.2f}')

... look at the output ...
>>>
```

One thing to emphasize here is that the class `Stock` acts like a factory for creating instances of objects. Basically, you call it as a function and it creates a new object for you. Also, it must be emphasized that each object is distinct—they each have their own data that is separate from other objects that have been created.

An object defined by a class is somewhat similar to a dictionary–just with somewhat different syntax. For example, instead of writing `s['name']` or `s['price']`, you now write `s.name` and `s.price`.

Exercise 4.2: Adding some Methods

With classes, you can attach functions to your objects. These are known as methods and are functions that operate on the data stored inside an object. Add a `cost()` and `sell()` method to your `Stock` object. They should work like this:

```
>>> import stock
>>> s = stock.Stock('GOOG', 100, 490.10)
>>> s.cost()
49010.0
>>> s.shares
100
>>> s.sell(25)
```

```
>>> s.shares
75
>>> s.cost()
36757.5
>>>
```

Exercise 4.3: Creating a list of instances

Try these steps to make a list of Stock instances from a list of dictionaries. Then compute the total cost:

```
>>> import fileparse
>>> with open('Data/portfolio.csv') as lines:
...     portdicts = fileparse.parse_csv(lines, select=['name','shares','price'], types=[str,
...
>>> portfolio = [ stock.Stock(d['name'], d['shares'], d['price']) for d in portdicts]
>>> portfolio
[<stock.Stock object at 0x10c9e2128>, <stock.Stock object at 0x10c9e2048>, <stock.Stock obje
 <stock.Stock object at 0x10c9e25f8>, <stock.Stock object at 0x10c9e2630>, <stock.Stock obje
 <stock.Stock object at 0x10ca6f7b8>]
>>> sum([s.cost() for s in portfolio])
44671.15
>>>
```

Exercise 4.4: Using your class

Modify the `read_portfolio()` function in the `report.py` program so that it reads a portfolio into a list of `Stock` instances as just shown in Exercise 4.3. Once you have done that, fix all of the code in `report.py` and `pcost.py` so that it works with `Stock` instances instead of dictionaries.

Hint: You should not have to make major changes to the code. You will mainly be changing dictionary access such as `s['shares']` into `s.shares`.

You should be able to run your functions the same as before:

```
>>> import pcost
>>> pcost.portfolio_cost('Data/portfolio.csv')
44671.15
>>> import report
>>> report.portfolio_report('Data/portfolio.csv', 'Data/prices.csv')
      Name     Shares      Price     Change
---------- ---------- ---------- ----------
```

```
        AA        100        9.22      -22.98
       IBM         50      106.28       15.18
       CAT        150       35.46      -47.98
      MSFT        200       20.89      -30.34
        GE         95       13.48      -26.89
      MSFT         50       20.89      -44.21
       IBM        100      106.28       35.84
>>>
```

Contents | Previous (3.6 Design discussion) | Next (4.2 Inheritance)
