Data Foundations: Objects

Instructor: Anthony Rios

Outline

Programming Patterns

We have discussed the 4 main programming patterns:

- Sequential Code
- Conditional Code (if, elif, else)
- Repetitive Code (loops)
- Store and Reuse (functions)

We also discussed basic data structures, like lists, sets and dictionaries.

Programming

Every programming tasks involves the use of **data structures** and writing **code that manipulates them**.

Elegance

There is always **multiple** ways to write a program.

However, some programs are “more elegant”, while other programs are “less elegant”

Object Oriented Programming

Object oriented programming is a way to arrange your code so that you can **zoom into 50 lines** of the code and **understand it** while **ignoring the other 999,950 lines** of code for the moment.

Object Oriented Programming

We have been using objects throughout this course!

```
>>> myList = []
```

is equivalent to...

```
>>> myList = list()
```

Object Oriented Programming

```
# This line CONSTRUCTS an object of type "list"
>>> stuff = list()

# The following lines CALL different METHODS
>>> stuff.append("python")

>>> stuff.append("chuck")

>>> stuff.sort()

>>> print(stuff[0])

>>> print(stuff.__getitem__(0))

>>> print(list.__getitem__(stuff, 0))
```

Methods vs Functions

A method is a **function** that is contained **within a class** and the objects that are constructed from the class.

An **object** contains both data and **methods** that manipulate that data.

- An object is active, not passive; it does things
- An object is responsible for its own data
    - But: it can expose that data to other objects

An object has state

An object contains both **data** and methods that manipulate that data

- The data represent the **state** of the object
- Data can also describe the relationships between this object and other objects
- Data is also known as “attributes”

Example: A “rabbit” object

Assume we want to create an object that represents a rabbit.

It would have **data**:

- How hungry it is
- How frightened it is
- Where it is

And **methods**:

- eat, hide, run, dig

Everything in Python is an Object!!!!

- type() – Returns the type of an object

- dir() – Returns the attributes and methods of an object

type()

type() – Returns the type of an object

```
>>> num = 3

>>> type(num)
int

>>> a = [1,2,3]

>>> type(a)
list

>>> type([1,2,3])
list
```

type()

```
>>> a = [1,2,3]

>>> b = [4,5,6]

>>> type(a) == type(b)
True

>>> c = 3

>>> type(a) == type(c)
False
```

dir()

dir() – Returns the attributes **and** methods of an object

```
>>> a = 3

>>> dir(a)
['__abs__',
'__add__',
'__and__',
'__bool__',
... ]
```

Exercise 1

This exercise has two parts. First, answer the following questions:

- What is the division **method** called for float objects?
- How many methods and attributes are there for float objects?
- How many methods and attributes are there for list objects?

Second, write code that uses the division **METHOD** (don't use "/", but the result should be the same as using "/") to divide a float object with the value 42 by 7.

Creating Objects in Python

example.py

```
class PartyAnimal:
        x = 0 # data/attribute
        def party(self): # Method
                self.x += 1
                print("So far", self.x)
```

Creating Objects in Python

example.py

```
class PartyAnimal:
        x = 0 # data/attribute
        def party(self): # Method
                self.x += 1
                print("So far", self.x)
an = PartyAnimal()
an.party()
an.party()
an.party()
```

```
anthony@MacBook:~$ python example.py
So far 1
So far 2
So far 3
```

Object life-cycle

example.py

```
class PartyAnimal:
        x = 0
        def __init__(self): # Called when object is created/initialized
                print('I am constructed')
        def party(self):
                self.x = self.x + 1
                print('So far',self.x)
        def __del__(self): # Called when object is destroyed
                print('I am destructed', self.x)
an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

Object life-cycle

example.py

```
...
an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

```
anthony@MacBook:~$ python example.py
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

Exercise 2

Write a class named "CheckingAccount" that contains the current **balance** of the account (an int) and the following methods:

- **__init__** – takes a "balance" parameter to initialize the data (balance) of the object.
- **withdraw** – takes an input parameter "amount" and modifies the data by reducing the balance. If "amount" results in an overdraw, subtract an extra 20 dollars. This method should return the balance balance.
- **deposit** – takes an input parameter "amount" and modifies the data by increasing the balance by "amount".

Write a few test cases to check your work.

Multiple Instances: Error on page 181 py4E

example.py

```
class PartyAnimal:
        x = 0
        name = ''
        def __init__(self, nam):
                self.name = nam
                print(self.name,'constructed')
        def party(self):
                self.x = self.x + 1
                print(self.name,'party count',self.x)
s = PartyAnimal('Sally')
j = PartyAnimal('Jim')
s.party()
j.party()
s.party()
```

```
anthony@MacBook:~$ python example.py
Sally constructed
Jim constructed
Sally party count 1
Jim party count 1
Sally party count 2
```

Inheritance

example.py

```
from party import PartyAnimal
class CricketFan(PartyAnimal):
        points = 0
        def six(self):
                self.points = self.points + 6
                self.party()
                print(self.name,"points",self.points)
s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))
```

Inheritance

example.py

```
s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
```

```
anthony@MacBook:~$ python example.py
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
```

Inheritance

Here we **override** the instructor of the original method __init__.

example.py

```
from party import PartyAnimal
class CricketFan(PartyAnimal):
        points = 0
        def __init__(self):
                PartyAnimal.__init__(self, “Anthony”)
        def six(self):
                self.points = self.points + 6
                self.party()
                print(self.name,”points”,self.points)
j = CricketFan() # No parameter needed now
```

```
anthony@MacBook:~$ python example.py
Anthony constructed
```

Exercise 3

Given the code for this exercise (already provided in notebook), answer the following questions in your notebook:

- What are the parent and child classes here?
- What does the code print out? (Try figuring it out without running it in Python)
- Which get description method is called when 'study spell(Confundo())' is executed? Why?
- What do we need to do so that 'print Accio()' will print the appropriate description ('This charm summons an object to the caster, potentially over a significant distance')? Change to code to do this.

Try to answer the questions before running the code!

Operator Overloading

Python operators (+, -, /, >, etc.) work for built-in classes.

But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

How can we use these operators with our own classes?

Special Functions

- __add__ (+) – Addition
- __sub__ (-) – Subtraction
- __eq__ (==) – Equality
- __gt__ (>) – Greater than
- __ge__ (>=) – Greater than or equal to
- __lt__ (<) – Less than
- __le__ (<=) – Less than or equal to
- __str__ – Python does not know how to print your object, thus if you print a new object, it will not be what you think. With this method you can tell Python how it should display your object.
- ...

Point Example

example.py

```
class Point:
        def __init__(self, x=0, y=0):
                self.x = x
                self.y = y
        def __str__(self):
                return "(0,1)".format(self.x, self.y)
```

Overloading the + Operator

example.py

```
class Point:
        def __init__(self, x=0, y=0):
                self.x = x
                self.y = y
        def __str__(self):
                return "(0,1)".format(self.x, self.y)
        def __add__(self, other):
                x = self.x + other.x
                y = self.y + other.y
                return Point(x, y)
```

Overloading the + Operator

```
>>> p1 = Point(1, 2)

>>> p2 = Point(3, 4)

>>> p3 = p1 + p2

>>> print(p3)
(4, 6)
```

Exercise 4

Expand on the Point class presented above by overloading the following operators:

- Equality (==/__eq__) – Two points should be equal if all the values are equal (i.e., x in self is equal to x in other)
    - This method should return either True or False
- Greater than (>/__gt__) – One point is greater than another if the magnitude of self is greater than other.
    - This method should return True or False.
    - Calculate magnitude as $x^2 + y^2$

The End

The End