EC2401: INTRODUCTION TO DATA SCIENCE

OBJECT ORIENTED PROGRAMMING

Evan Piermont

Problem of the Week

Aileen, Beatrix, and Cecelia decided to play ping pong. Only two people can play at a time, so they agree that two of them will start playing, and then whoever loses the game will swap out with the player who was sitting out. Aileen played 17 games, Beatrix played 10 games, and Cecelia played 15 games.

Who lost the second game?

Types

Recall, a any data that python works with has a type is a program that takes inputs and returns outputs.

- `True` and `False` Boolean
- `0`, `1` `2` are integers
- `'abc'` `'Hello World!'` are strings
- `[x,y,z, ...]` is a list
- `print`, `map`, `sum` are functions

These objects have built in functions (and attributes) attached to them. For example:

- `list.append()` is a function that is specific to lists, and adds a new element
- `list.reverse()` is a function that is reverses the order of the list
- `list.pop()` is a function that removes (and returns) and element

Where do these functions come from? What if we want to define our own data structures?

Object Oriented Programing

Object Oriented Programming (OOP) is a widely used programming paradigm built from:

- Objects
  - A specific instance of a pattern, like a real world object
- Classes
  - prototype or blueprint from which objects are created
  - Types and Classes are the same thing for our purposes

- A **Class** (for example `List`) has
  - A set of attributes or properties that describes every object (e.g., a lists length: len)
  - A set of methods (functions) that every object can perform (e.g., append)
- An **Object** (for example `lst= [0,1,2]`) has
  - A set of data (e.g., its length is `len(lst) = 3`)
  - A set of actions that it can perform (e.g., `lst.append(3)`)
  - An identity (location in memory, bound to `lst`)

We can create our own classes:

```
class newClass:
    def __init__(self, var):
        self.attr1 = CODE, DEPENDS ON var
        self.attr2 = CODE, DEPENDS ON var

instance = newClass(x)
```

Lets unpack this...

```
class newClass:
    def __init__(self, var):
        self.attr1 = CODE, DEPENDS ON var

instance = newClass(x)
```

- The last line will create an instance of `newClass`, setting `var = x`
- `__init__` is a special function, that initializes, or creates an instance: it is what is called by the last line
- Attributes are saved in `self.attr` and can be assessed in the same way
- `self` is a special var that refers the the instance itself, it does not need to be passed when functions are called

```
class newClass:
    def __init__(self, var):
        self.attr1 = CODE, DEPENDS var

    def newMethod(self,vars):
        FUNCTION INNER
        return out

instance = newClass(x)
instance.method(y)
```

- We can put function definitions inside the class constructor, these are bound to the instances
- We can then call these functions (without needing to pass `self`, as with `__init__`) by `instance.newMethod()`

Why OOP: Abstraction

Classes represent the essential properties and behavior of an entity.

- Highlights the most important facets of the data
- Solves problems using these general features: reduces need for case-by-case handling

Why OOP: Encapsulation

Classes combine data and the functions that manipulate the data into one unit

- Conceptually cleaner, keeps everything in one place
- Forces scope, so we know where functions apply
- Thinks of methods/functions as data themselves

Why OOP: Inheritance

Derived classes, classes that are specifications of existing classes (base classes) inherits the attributes and methods of the base class

- Further abstraction, allows for general and specific cases
- Minimizes redundant code, allows for propagation

```
class baseClass:
    def __init__(self, var):
        self.attr1 = CODE, DEPENDS ON var

    def newMethod(self,vars):
        FUNCTION INNER
        return out

class derivedClass(baseClass):
    def __init__(self, var, moreVar):
        baseClass.__init__(self, var)
        self.attr2 = CODE, DEPENDS ON var, moreVar

instance = derivedClass(x,y)
```

```
class derivedClass(baseClass):
    def __init__(self, var, moreVar):
        baseClass.__init__(self, var)
        self.attr2 = CODE, DEPENDS ON var, moreVar

drvdInstance = derivedClass(x,y)
```

- The object `drvdInstance` is of class `derivedClass` which is derived from `baseClass`
- It therefore has all the attributes and methods of `baseClass` , like `drvdInstance.attr1` or `drvdInstance.newMethod()`
- In addition it has its own attributes, like `drvdInstance.attr2`, that `baseClass` does not have

Disadvantages of OOP

- Steep expectation to learn and adapt
- Bigger program size: OO programs commonly include more lines of code than procedural/functional
- Slower runtime: OO programs are normally slower than procedure-based programs, as they ordinarily require more guidelines to be executed.
- Not appropriate for a wide range of issues.

Here is a link to live examples.