Python Object-Oriented Programming
Software Engineering for Scientists

What is Object-Oriented Programming?

You can think of OOP as programming based on smaller objects bundled together to construct a program. The properties and behaviors of your program live *inside* these objects.

What is Procedural Programming?

In contrast, procedural programming is like a recipe with steps you follow in sequence. In procedural programming, we have functions structured in blocks of code and the behavior of the program flows in sequence from one block to another.

Scenario: We have a data file with information on cell types, and how old each cell is. We want to analyze this file and add various cells together.

OOP: We create a class called `Cell`. This class has variables type and age that will describe a specific cell and the method `mitosis()` which divides will divide a cell into two cells. When we want to create many cells from the information in the file, we create a unique instance of the class, called an *object* inside the main function. When we want to divide a cell, we call the `mitosis()` function of that particular cell.

PP: We create a method to read in the file. We take these results and create cells. In order to divide cells, we create a `mitosis()` function that can divide a cell. The functions are not inherent to a cell so a cell cannot choose to divide itself.

Why do we want to use this?

OOP breaks down your program into smaller, more manageable pieces. This helps control complexity in large software systems, makes the code easier to read, and lets you easily reuse pieces in other projects. Plus, the Python modules you have already been using invoke OOP and you should understand how they work.

Vocab

Class: the *template* for what the object is and what it can do
Object: the *specific instance* of a class. When you create a new object from a class, you *instantiate* it.
Method: a function within a class
Attribute: a variable within a class that is specific to that class
Private: a variable or function that is only accessible within a class
Public: a variable or function that things outside the class can access

*Note that Python does not really have notions of private and public, the convention for private is to use an underscore when naming the function or variable to signal to others it is private. We will not worry much about this.

Four Principles

Encapsulation: Encapsulation means making methods and attributes inside an object accessible to only that specific instance of the object. You can have many different types of cells in your code, each cell divides differently and some have functions others do not. Encapsulation happens when each object keeps its

state private. For example: Cell A cannot divide Cell B if the method `mitosis()` is private, but there could be another method `signal_to_divide()` that Cell A could use to signal Cell B to divide.

Abstraction: An extension of encapsulation. You "abstract away" the details of a class so it can do many actions "under the hood" but other parts of your code only need to interact with functionality. Abstraction is a very broad concept, but it essentially involves hiding unnecessary details and building complexity one level at a time.

Inheritance: You can build classes on top of other classes! You can think of this as levels, for example maybe there is a general class `animal` with variables and functions. Then you want to create a type of animal – a `dog`, the dog is an animal, so it can inherit all the functions inside the animal class, plus some new dog-specific variables and functions. Inheritance can be as many levels as desired. This promotes reuseability: the same animal class can be inherited by dogs, cats, rabbits etc without having to rewrite.

Polymorphism: This means that an object or function can respond differently based on the class. For example, you might have a function called `divide()` inside the class cell which is called on an object of type cell and signals the cell to divide (eg multiply), or you might have the function `divide()` inside a class of math functions that is called on integer values and divides them.

Ok, how do we use this in Python?

First, let's talk about how you structure your file, the order should *always* be:
Top: import statements
Middle: classes, functions
Bottom: main function

By following this format it makes it very easy for others to follow your code. If you want to know what the code will do, you find the main function, **at the bottom**.

`cell_lib.py`

```
import argparse
import math_lib

class Cell:
    def __init__(self, source, age):
        self.source = source
        self.age = age

    def mitosis(self):

if __name__ == '__main__':
    cell = Cell()
```

Now, let's look at how you start a class:
We use the keyword, class. Everything that lives inside the class is tabbed. The name of the class uses CamelCaseForEachWord.

You can choose to create class attributes which are variables that will be constant for EVERY instance of that class. Do this first.

Then we create a special function called the initializer. The syntax is
`def __init__(self, inputs):`

```
class Cell:
    # class attributes
    TYPE = 'human'

    # Initializer / Instance attributes
    def __init__(self, source, age):
        self.source = source
        self.age = age
```

This is the function that is called when you instantiate a new object of that class.

The keyword we use is self. This is a style guide. When we write other functions inside the class, we also use self.

The dot operator & self

- The dot operator `(.)` is how we say something "lives inside" something else. For example, the attribute source is part of the cell instance we will create.
- When we call the method, the self is not included as in input argument

We can also create variables that are only helper variables, not attributes of the class. These **do not** have the `self.` prefix

```
class Cell:
    # class attributes
    TYPE = 'human'

    # Initializer / Instance attributes
     def __init__(self, source, age):
         self.source = source
         self.age = age
         helper = []
```

Scope

The scope of variables with the prefix self is throughout the class. However, if you create helper variables inside functions, those variables exist only inside that function.

In most cases we will use the keyword `self` and create instance methods. These methods can change the state of a specific instance of a class (aka an object). They have access to variables inside the cell that have the prefix `self`. These methods can have no arguments (apart from `self`) or they can have an unlimited number of input arguments. They can have a `return` statement, or not.

```
class Cell:
   # class attributes
   TYPE = 'human'

   # Initializer / Instance attributes
    def __init__(self, source, age):
        self.source = source
        self.age = age
        helper = []

   def grow_year(self):
        self.age += 1
```

How do we call this method? With the dot operator!

```
>> CellA = Cell('nerve', 3)
>> CellA.age
3
>> CellA.grow_year()
>> CellA.age
4
```

What is a class method?
In some cases, you may see the keyword `cls`. This is to be *used only for class methods* and the class method should be flagged with @classmethod

A class method only has access to the cls parameter point to the class and can only modify the *class state* that is consistent across all instances of the class. It cannot modify an object's instance state (that requires the self parameter). You can use this to create multiple objects of the class that all configured in some specific way.

The most practical use of a class method is to have a class method that creates an instance of that class with specific inputs.

```
class Cell:
   # class attributes
   TYPE = 'human'

   # Initializer / Instance attributes
   def __init__(self, source, age):
        self.source = source
        self.age = age

   def grow_year(self):
        self.age += 1

   @classmethod
```

```
    def create_epithelial_cell (cls):
        return cls('epithelial',0)
```

How do we call this method? The dot operator!

```
>> Cell.create_epithelial_cell()
Cell('epithelial',0)
```

Inheritance

Inheritance provides a nice way to construct multiple related classes (think `dog, elephant, mouse` which are all part of an overarching class of `animals`), or build complexity in levels (think `cell` → `organ` → `person`). We can easily construct a class that "inherits" methods and attributes of another class. This new class is referred to as the *child* and the original class is the *parent.*

In order to say a class is built on another, we pass the parent class name to the child class when we define it:

```
class Cell:
    # class attributes
    TYPE = 'human'

    # Initializer / Instance attributes
    def __init__(self, source, age):
        self.source = source
        self.age = age

    def grow_year(self):
        self.age += 1

class Organ(Cell):
    def __init__(self, name):
        self.name = name
```

Note: the child's `__init__` function will override that of the parent.
You can keep the parent's __init__ function by calling it:

```
class Cell:
    # class attributes
    TYPE = 'human'

    # Initializer / Instance attributes
    def __init__(self, source, age):
        self.source = source
        self.age = age

    def grow_year(self):
        self.age += 1
```

```
class Organ(Cell):
    def __init__(self, name,source,age):
        self.name = name
        Cell.__init__(self,source,age)
```

You can also use the super() function as a proxy for the parent class:

```
class Cell:
    # class attributes
    TYPE = 'human'

    # Initializer / Instance attributes
    def __init__(self, source, age):
        self.source = source
        self.age = age

    def grow_year(self):
        self.age += 1

class Organ(Cell):
    def __init__(self, name, source, age):
        super().__init__(self, source, age)
```

Now that we have constructed a class, how can instantiate objects of the class?

- With the dot operator!
- But we do not use the keyword self or cls in the input arguments

```
class Cell:
    # class attributes
    TYPE = 'human'

    # Initializer / Instance attributes
    def __init__(self, source, age):
        self.source = source
        self.age = age

    def grow_year(self):
        self.age += 1

    @classmethod
    def create_epithelial_cell (cls):
        return cls('epithelial',0)
```

```
if __name__ == '__main__':
    cell1 = Cell('nerve', 0)
    cell2 = Cell('epithelial',4)

    cell1.grow_year()
```