Python Session 4

After going through the content, please take a minute to respond to this feedback survey of our reference manuals.
Feel free to use the outline on the left side of the document to jump to different sections.
Classes are also usually a difficult concept to understand. If you have any questions, feel free to come to the office hours hosted by Sahas and Scott!

Classes

What are classes?

Classes are extremely useful in Python and in any programming language in general. In programming, it can be useful to put related variables and functions together in a class to serve as a blueprint for future uses where you may want to use the same groupings.

Object Oriented Programming

Classes are the main features in **object oriented programming**, which is extremely important in the world of programming. **Classes** are the blueprints while **objects** are the actual groups that are created from the blueprints. Multiple objects can be created from the same class blueprint. In each blueprint, there are variables and functions called **member variables** and **member functions**, respectively. Every object is independent from one another; any changes to one object will not affect the other.

```
class car:
  def __init__(self, wheels):
    self.wheels = wheels

f = car(2)
d = car(4)              # multiple cars can be created from the car class blueprint
print(f.wheels)
f.wheels = 5            # changing f's wheels does not change d's wheels
print(d.wheels)
```

```
2
4
```

Class Definition

Classes must be defined before you can use them. To define them, type `class <class_name>:`. **All variable and function declarations indented after this will be a part of the class**. Unlike in other programming languages, you don't need to pre-define the member variables to say that it is a member of that class. They can be created in a function or in the constructor to become a member of the class. Both of the following class variations do the same thing: they declare a **`wheels`** member variable in the class that has a default value of **`0`**.

```
class car:
  wheels = 0

class car:
  def __init__(self, wheels = 0):
    self.wheels = wheels
```

Any member variable created in a member function needs a **`self.`** prepended to it (**`self.<attribute_name>`**); otherwise, the variable will only be available for that function and not the entire class. Member variables created outside a member function **do not need** to prepend the **`self.`**
Similarly, to get access to any class attributes in a member 1, you must prepend **`self.`** when you call them.

Member functions are created like any other function, except it needs to be within the class to be a member function. It is important to note that member functions **always** have a **`self`** parameter that points to the class it is within. This allows the function to access the different **attributes** (or the member variables and member functions) of the class. The argument for the **`self`** parameter is **implicit**, and does not need to be passed in manually.

```
class car:
  wheels = 4                    # self. NOT needed as it's created in the class
  def print_wheels(self):
    print(self.wheels)          # self. needed to access wheels

class car:
  def __init__(self, wheels = 0):
    self.wheels = wheels        # self. NEEDED to make wheels a member of the class
    # wheels did not exist before, self. creates and assigns it to the class
```

Class Constructor

Classes normally have a **constructor**. Constructors are functions that you can create in a class that will be **automatically called when creating a new object**. Usually, constructors will set-up the class. They may have parameters that can be passed in to use to assign any member variables with.

Constructors are defined as **`def __init__(self, <parameters>):`**.Any member variables defined in the constructor must have **`self.`** prepended to it so that it can be used in other functions.

```
class car:
  def __init__(self, wheels):         # constructor that requires a parameter wheels
    self.wheels = wheels

f = car(2)              # 2 passed in for wheels, self is implicitly passed in
print(f.wheels)
```

```
2
```

Objects and Calling

Objects are essentially custom variables. Objects are created like this: **`<obj_name> = <class>()`**. For example:

```
f = car()
```

`f` is now a **`car`** object based on the **`car`** class blueprint. The object name goes on the left of the assignment operator, with the right side being the class followed by parentheses. The parentheses is where any arguments are passed into if the constructor requires it. **Note that `self` does not need an argument passed in as it is implicitly passed in**.

```
...             # using the car class defined earlier
f = car(2)      # 2 is passed in for wheels, self does not need an argument
```

An object's attributes can be called with the **dot operator** following the object's name, with the attribute to be called following the dot (**`<obj_name>.<attribute_name>`**) like in the following example:

```
...             # using the object declared in the previous code box
print(f.wheels) # prints out the wheels member variable in object f
```

An object's attributes are by default **public** to the program after its declaration if the variable and functions are declared with normal naming schemes.

Private Members

Sometimes, we want **private members** that are inaccessible by the user outside of the class. To make an attribute private, **prepend at least 2 underscores onto the attribute's name**. Member functions of that class will be able to use it, but it cannot be called outside the class.

⚠ **Appending 2 or more underscores onto a private attribute will nullify the private effect.**

```
class car:
  __brand = 3              # creates a private member variable
  __model__ = "default"    # still public as the __ after removes the private effect
  def __init__(self, wheels, brand="default"):
    self.wheels = wheels
    self.__brand = brand   # member functions can use the private variable

f = car(4, "Toyota")
print(f.wheels)
print(f.__model__)   # __model__ is not private, we can access it
print(f.__brand)     # __brand is private, it gives us an error
```

```
2
```

Extras

Objects in Classes

We put this as an extra as it is not really an emphasis in the lecture; however, it is important to know this information, especially for the assignment. Like any variable, objects of other classes can also be used within a class. To do this, it will be as if you declared a normal object in the class instead of globally.Likewise, you can pass objects through functions to be assigned/copied into a class object.

```
class Note:
  text = "Hello"

class Notebook:
  note = Note()               # object within another class
  def print_notebook(self):
    print(self.note.text)

notebook = Notebook()
notebook.print_notebook()
```

```
Hello
```

C++ and Python

Python's classes are different from classes in *C++*. If you are familiar with *C++*, Python's classes are equivalent to structs more than the actual classes. For those unfamiliar with *C++*, classes in that programming language are by default private, and public and private attributes are grouped together within the class rather than assigned.
Python:

```
class car:
  wheels = 0
  __brand = "Toyota"
```

C++ equivalent:

```
struct car:
  wheels = 0;
private:
  brand = "Toyota";
```

Similarly, the **`self`** parameter is exactly the same as if you used **`this`** in *C++*; although, **`this`** is not a parameter, and rather *C++* already considers it as a keyword within the class that points back to itself.