

## CC3 Object Oriented Programming Midterm Notes

### I. CLASSES AND OBJECTS IN PYTHON

#### Creating Python Classes

- Creating a class in Python work similarly to declaring a class
- The syntax for the class in Python 3 looks like this:

```
class MyFirstClass:
    pass
```
- The class definition starts with the "class" keyword
- This is followed by a name identifying the class and is terminated with a colon
- The class name follows the standard Python variable naming rules
- The class definition line is followed by the class contents which is indented
- Like functions, indentation is used to delimit the classes
- The class cannot be empty, so if you would like to create placeholder content, you can make use of the "pass" keyword
- Creating a class allows us to instantiate objects of that class

#### Adding Attributes

- Now that you have a class, you can start defining the attributes for the class
- If you recall, the attributes are the characteristics that we can use to describe the class
- There are two ways that we can define the attributes for the class
- The first method is to define the attributes on an instantiated object using the dot notation:

```
class Point:
    pass
var = Point()
var.x = 5
var.y = 4
```
- This allows us to assign the "x" and "y" attributes to the "Point" class
- This method creates an empty "Point" class with no data or behaviors
- Then it creates two instances of that class and assigns each of those instances "x" and "y" coordinates
- We can assign a value to an attribute on an object with the following syntax  
`<object>.<attribute> = <value>`
- This is sometimes referred to as a dot notation
- The value can be anything:
  - A Python primitive
  - A built-in data type
  - Another object
  - A function
  - A class
- The second method is to define the attributes inside the class and then call the attribute using the dot notation:

```
class Person():
    species = "Homo Sapiens"
var = Person ()
```
- In this method, we define attributes are normal variables
- This method creates a "Person" class with the attribute "species" is given the value of "Homo Sapiens"
- The attributes "species" is given the value of "Homo Sapiens"
- We can assign a value to an attribute inside the class with the following syntax:  
`<attribute> = <value>`
- This allows us to declare the attribute inside the class itself
- We can get the value of the attribute with the dot notation
- This type of attributes applies to the whole class and are called class attributes
- This means that when you define a new object with the class, the object will have all the attributes defined inside the class

## Adding Behaviors

- Now that we have our attributes, we can then implement the behaviors of our class
- To define behaviors in our class, we make use of functions created inside the class
- The syntax of creating methods or behaviors inside our class are as follows:

```
class ClassName:
    def method_name(self):
        pass
```

- Let us use the "point" class from earlier for our example
- We can create a "reset" method that moves the point to the origin (a state where "x" and "y" are both zero)

```
class Point:
    def reset(self):
        self.y = 0
```

```
p = Point()
p.reset() #outputs 0
self.x = 0
```

- As mentioned earlier, a method in Python is formatted identically to a function
- It starts with the keyword "def" followed by a space and the name of the method
- This is followed by a **set of parameters** containing the parameter list terminated with a **colon**
- The next line are the statements inside the method

## The "self" argument

- The one difference between methods and normal functions is that all methods have one required argument
- This argument is conventionally name "self", take note that this can be named anything else but for consistency, let's use "self".
- The "self" argument to a method is simply a reference to the object that the method is **being invoked on**
- We can access attributes and methods of that object as if it were any other object

- When calling a method, we do not have to pass the "self" argument into it
- Python automatically takes care of this for us
- It knows we're calling a method to a specific object, so it automatically passes that object to the method
- This is done because in Python, we have methods that make the instance to be passed automatically, but not received automatically

- To help us understand the "self" argument, let us call a method in a different way
- Instead of using the syntax <object>.<method> we can instead call a method in another manner:

```
object = Class()
Class.method (object)
```

- This is alternative format that we can use for calling a method in a class

- If we were to apply this to the "Point" example that we have been using earlier, our code will look like this:

```
p = Point()
Point.reset(p)
print(p.x, p.y)
```

- Here, we can see that we are passing the object "p" as our argument when calling the "reset" method
- This is the object that is being passed to the "self" argument

If we do not include the self-argument in our class definition, Python will give us an error message:

```
Traceback (most recent call last):
  File
C:\Users\ibrah\PycharmProjects\pythonProject\main.py
",line 35, in <module>
    p.reset()
TypeError: Point.reset() takes 0 positional
arguments
but 1 was given
```

## Using Multiple Arguments and Methods

- Classes are not limited to just one method and each method is not just limited to one argument
- We can add additional methods to our class which is then usable by any objects that are instantiated for that class
- This allows us to create classes with many methods and behaviors that manipulate the attributes stored in them
- Let us use our previous class "Point" as an example, we shall expand the number of methods and arguments within the class:

```
import math

class Point:
    def move(self, x, y):
        self.x = x
        self.y = y
```

- Let us use our previous class "Point" as an example, we shall expand the number of methods and arguments with the class:

```
def reset(self):
    self.move(0, 0)

def calculate_distance(self, other_point):
    return math.sqrt(
        (self.x - other_point.x) **2 + (self.y - other_point.y) **2
    )
```

- Let us use our previous class "Point" as an example, we shall expand the number of methods and arguments within the class:

```
point1 = Point()
point2 = Point()

point1.reset()
point2.move(5, 0)
print(point2.calculate_distance(point1))

point1.move(3, 4)
print(point1.calculate_distance(point2))
print(point1.calculate_distance(point1))
```

- As seen in the previous example, the class "Point" now has three methods
- The "move" method accept two arguments, "x" and "y", and sets the values on the "self" object
- The "reset" method now calls the "move" method, since this method is just moving to specific coordinates
- The "calculate\_distance" method uses Pythagorean theorem to calculate the distance between two points

## Initializing Objects

- Depending on what type of class you are creating, you might want to initialize the attributes of your class
- If you would like to initialize the values for your object, there are two ways to accomplish this
  - The first method is the simply set a value directly for the attributes of your class
  - The second method is to make use of an OOP concept called a constructor
    - In Python, the language takes this a step further by having an initializer and constructor
- In Python, there is a special method that works as an **initializer** and allows us to set default values for the class
- The Python **initialization method** works the same as the other methods, except it has a special name:

```
def __init__()
```
- The leading and trailing double underscores means that this is a **special method** that Python will treat as a special case

- Let's use our example of the "Point" class and add an initializer to it

```
class Point:
    def __init__(self, x, y):
        self.move(x, y)
    def move(self, x, y):
        self.x = x
        self.y = y
    def reset(self):
        self.move(0, 0)
point = Point(3, 5)
print(point.x, point.y)
```

- In our example, we can now construct a point when an object is initialized
- This ensures that an error will not occur when the "x" and "y" attributes are called
- Take note that we will have to make sure that we provide all the arguments requested in the "\_\_init\_\_" method to avoid errors
- Just like functions, we can also set default values in the "\_\_init\_\_" method
- This will allow us to accept incomplete or no arguments in our class when an object is initialized
- This can be done similarly to the example below:

```
class Point:
    def __init__(self, x=0, y=0):
        self.move(x, y)
```

- Aside from the initializer, we also have a constructor method in Python which is named the "\_\_new\_\_" method
- This accepts one argument, the class that is being constructed
- It returns the newly created object
- The syntax for this is shown below:

```
class class_name:
    def __new__(cls, *args, **kwargs):
        print()
        return super(class_name, cls).__new__(cls, *args,
**kwargs)
```

- An example of how the "\_\_new\_\_" method is used is shown below:

```
class NewClass(object):
    def __new__(cls):
        num1 = int(input("Enter the first number:"))
        num2 = int(input("Enter the second number:"))
        sum = num1 + num2
        print("The sum of the two numbers is", sum)

        return super(NewClass, cls).__new__(cls)
NewClass()
```

- As seen in this example, when the class is called, the program asks for two inputs for two numbers
- After the user inputs the values, the application then computes for the sum and prints it
- While this method allows us to override "\_\_new\_\_", in general, you shouldn't need to do this unless you're subclassing an immutable type like **str, int, Unicode or tuple**
- In practice, you will rarely, if ever need to use "\_\_new\_\_" and "\_\_init\_\_" will be sufficient

## Expanded Comments

- In the previous topics, you learned that you could write comments in Python with "#" symbol
- While this is useful for short comments, longer explanations can be harder to write with this method
- This is even more important in OOP, were documenting how a class and its methods works allows yourself and others to understand how to use it in your own applications
- This can be done with the use of **docstrings**
- Docstring are Python strings enclosed with an **apostrophe(') or quote(")**
- They span multiple lines, which can be formatted as multiline strings, enclosed in matching **triple('') or triple quote(''')** characters
- Although it does not have a hard limit, try to limit your explanation to 80 characters

- A docstring should clearly and concisely summarize the purpose of the class or method it is describing
- It should explain any parameters whose usage is not immediately obvious and is also a good place to include short examples of how to use the class and its methods
- Any caveats or problems an unsuspecting user of the class and its methods should be aware of also be noted
- An example of how docstring can be used is shown below:

```
class Point:
    'Represents a point in two-dimensional geometric coordinates'

    def __init__(self, x=0, y=0):
        '''Initialize the position of a new point. The x and y
        coordinates can be specified. If they are not, the point of defaults
        to the origin.'''
        self.move(x, y)

    def move(self, x, y):
        'Move the point to a new location in 2D space.'
        self.x = x
        self.y = y
```

## II. INHERITANCE IN PYTHON

### Introducing Inheritance

- To recall what this term means, inheritance allows us to create "is a" relationship between two or more classes
- This allows us to abstract common logic into super-classes and manage specific details in the subclass
- The superclass is the "parent" that has all the methods and attributes
- The subclass is the "child" that inherits all the methods and attributes of the superclass
- Technically, every class we create in Python uses inheritance
- All Python classes are subclasses of the special class named object
- This class does not really do much in terms of attributes and behaviors, but it does allow Python to treat all object in the same way
- If we don't explicitly inherit from a different class, our classes will automatically inherit from object
- A new class can be derived from an existing class as follows:
 

```
class MyClass(object):
    pass
```
- In Python 2.2 and Python 3 onwards, all classes implicitly inherit from the object class
- This means we do not need to type object when defining the parent class
- This was just show to demonstrate how Python tells classes they can be inherited
- A new class can be derived from an existing class as follows:
 

```
class SuperClass(object):
    pass
class SubClass(SuperClass):
    pass
```
- This syntax tells Python that the "SubClass" should be derived from the "SuperClass"
- Now that we know how to define subclass and superclass, let us see how it can be used
- The most straightforward use of inheritance is to add functionality to an existing class
- Let us have some examples of how this can be used on the next slides

### Using Inheritance

- Here is a simple example of how inheritance can be implemented:

```
class Cat:
    def __init__(self, breed, color):
        self.breed = breed
        self.fur_color = color
    def meow(self):
        print("The cat meows.")
```

- Here is a simple example of how inheritance can be implemented:

```
class PersianCat(Cat):
    pass
persian_cat1 = PersianCat("Persian Cat", "White")
print(persian_cat1.fur_color)
persian_cat1.meow()
```

- As seen in the previous example, the "PersianCat()" class inherits the attributes and methods of the "Cat()" class
- This also includes "\_\_init\_\_" method, which means the child class will also need to instantiate an object
- When we would like to use the attributes in the "Cat()" class, we simply need to call them
- We do not need to define the same attributes and methods in our child class "PersianCat()"

### The super() function

- The super function returns the object as an instance of the parent class, which allows is to call the parent method directly
- This essentially allows a child class to call all the methods of the parent class
- This is meant to allow code reusability
- This allows you to override methods in your child class, but still access the "original" methods in your parent class

### Using Inheritance

- We define a super class "Rectangle" with the following attributes and behaviors:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    def perimeter(self):
        return 2 * (self.width * self.height)
```

- We can now define a subclass "Square"
- Since a square is simply a rectangle with equal sides, these two shapes share characteristics and can be inherited
- Here is the source code example for the subclass "Square":

```
class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)
        self.side = side
```

- We use the "super()" method to allow us to set the sides of the square with the help of the parent class
- What is happening is we are asking the user to give a value for the square when it is instantiated
- We then pass the value as both the height and width to the "Rectangle" class initializer
- This allows us to set the sides of the squares to be equal

- Once you have your superclass and subclass linked together, you can then start creating objects for both classes
- Keep in mind that objects created for their class are independent of each other
- Here are some examples of objects created along with their outputs:

```
rectangle_example = Rectangle(2, 3)
print(rectangle_example.area())
print(rectangle_example.perimeter())
```

- We can now also create objects for our "Square()" class by passing only one value:

```
square_example = Square.(5)
print(square_example.area())
print(square_example.perimeter())
```

## Overriding Class Behavior

- Inheritance allows us not to only add new behavior to existing classes but change behavior
- Overriding means **altering or replacing** a method of the superclass with a new method in the subclass
- When overriding methods, you are usually only overriding methods of the **same name**
- No special syntax is needed to do this
- You simply need to recreate the method you would like to override
- The **subclass's newly created method is automatically** called instead of the superclass's method