CS: Objects in Python

Introduction to Objects

Classes and Objects

Learning Objectives - Classes and Objects

- **Define the terms class, objects, instance, and instantiate**
- **Identify the difference between classes and objects**
- **Create a user-defined object (with `pass` as the body)**
- **Add an attribute to an object with dot notation**
- **Define class attribute**
- **Explain the difference between shallow and deep copies**

Built-in Objects

The String Object

You have already been using built-in Python objects. Strings are an example of a Python object.

```
s = "I am a string"
print(type(s))
```

Python says that the type of `s` is the class `str` (which is a string). Add the following line of code and run the program again.

```
print(dir(str))
```

`dir` is a built-in function that tells the user all of the attributes and methods associated with a class. If you look carefully at the output, you may be confused by the information on the screen. However, a few things, like `upper` and `lower`, may seem familiar. Attributes and methods will be covered in a later lesson, but it is important to understand that a string is not a simple collection of characters. Because a string is a class, it is a powerful way of collecting and modifying data.

challenge

What happens if you:

- Add the line of code `print(s.__add__("!"))`?
- Add the line of code `print(s.__sizeof__())`?
- Add the line of code `print(s.startswith("I"))`?

Vocabulary

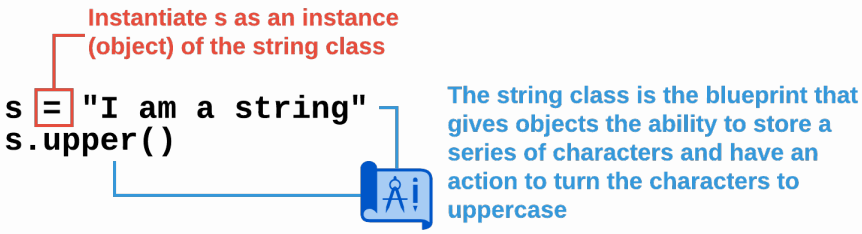
In the text above, the words “class” and “object” are used in an almost interchangeable manner. There are many similarities between classes and objects, but there is also an important difference. Working with objects has a lot of specialized vocabulary.

Classes - Classes are a collection of data and the actions that can modify the data. Programming is a very abstract task. Classes were created to give users a mental model of how to think about data in a more concrete way. Classes act as the blueprint. They tell Python what data is collected and how it can be modified.

Objects - Objects are constructed according to the blueprint that is the class. In the code above, the variable `s` is a string object. It is not the class. The string class tells Python that `s` has methods like `__add__`, `__sizeof__`, and `startswith`. When a programmer wants to use a class, they create an object.

Instance - Another way that programmers talk about objects is to say that an object is an instance of a particular class. For example, `s` is an instance of the string class.

Instantiation - Instantiation is the process where an object is created according to blueprint of the class. The phrase “define a variable” means to create a variable. The variable is given a name and a value. Once it has been defined, you can use the variable. With objects, you use the phrase “instantiate an object”. That means to create an object, give it a name, store any data, and define the actions the object can perform.



Class vs Object

User-Defined Objects

Defining an Object

Assume you want to collect information about actors. Creating a class is a good way to keep this data organized. The `class` keyword is used to define a class. For now, use `pass` as the body of the class. Declaring a class does not doing anything on its own, so print `Actor`.

```
class Actor:
    pass

print(Actor)
```

▼ **Naming classes**

The convention for naming classes in Python is to use a capital letter. A lowercase letter will not cause an error message, but it is not considered to be “correct”. If a class has a name with multiple words, do not use an _ to separate the words like for variable and function names. Instead, all of the words are pushed together, and a capital letter is used for the first letter of each word. This is called camel case.

Classes are just the blueprint. To you use a class, you need to instantiate an object. Here is an object to represent Helen Mirren.

```
class Actor:
    pass

helen = Actor()
print(type(helen))
```

So you now have `helen`, which is an instance of the `Actor` class.

Adding Attributes

The point of having a class is to collect information and define actions that can modify the data. The `Actor` class should contain things like the name of the actor, notable films, awards they have won, etc. These pieces of information related to a class are called attributes. You access the attributes by using dot notation, which is `object_name.attribute`. Adding an attribute is done with the assignment operator, `object_name.attribute = attribute_value`. Object attributes can be treated like any other variable.

```
class Actor:
    pass

helen = Actor()
helen.first_name = "Helen"
helen.last_name = "Mirren"
print(helen.first_name, helen.last_name)
```

challenge

What happens if you:

- Add the print statement `print(helen.first_name.upper(), helen.last_name.lower())`?
- Add the attribute `total_films` with the value of `80`?
- Add the attribute `notable_films` with the value of `["The Queen", "The Madness of King George", "Gosford Park"]`?
- Add the print statement `print(helen)`?

▼ **What is all that gibberish after `at`?**

All of that gibberish is the location in your computer's memory where the object is being stored.

The Constructor

Too Much Code

Using the `Actor` class from before, create a detailed object for Helen Mirren. Adding each attribute individually would require a lot of code.

```
class Actor:
    """Define the actor class"""
    pass

helen = Actor()
helen.first_name = "Helen"
helen.last_name = "Mirren"
helen.birthday = "July 26"
helen.total_films = 80
helen.oscar_nominations = 4
helen.oscar_wins = 1
print(helen.first_name, helen.last_name)
```

The class `Actor` does nothing but create a class. It does not create any attributes; the user has to do this. A class is suppose to be a blueprint. It should lay out all of the attributes for the user. Classes can do this when you use the constructor.

The Constructor

The constructor is a special method (more on methods in another lesson) for a class. Its job is to define all of attributes associated with the object. These attributes are called instance variables. In Python, the constructor is coded as `__init__`. The double underscores before and after `init` are required. The constructor also **must** have `self` as a parameter. Use the same syntax for the constructor as you would a function. Be sure to indent the constructor four spaces since it is a part of the class. Attributes will be referred to as `self.attribute_name`. Instantiating `helen` as an instance of the `Actor` class automatically calls the constructor.

▼ **What's up with the double underscores?**

Objects often have methods with double underscores. These are referred to as "dunder" methods. A dunder method is an implicit method and is called behind the scenes by an explicit operation. You never have to call `helen.__init__`. When you instantiate the object, Python calls **`init`** automatically.

```
class Actor:
    """Define the actor class"""
    def __init__(self):
        self.first_name = "Helen"
        self.last_name = "Mirren"
        self.birthday = "July 26"
        self.total_films = 80
        self.oscar_nominations = 4
        self.oscar_wins = 1

helen = Actor()
print(helen.first_name, helen.last_name)
```

challenge

What happens if you:

- Add the line of code `print("{} {}'s birthday is {}.".format(helen.first_name, helen.last_name, helen.birthday))`?
- Add the line of code `print("{} {} won an oscar {}% of the time.".format(helen.first_name, helen.last_name, helen.oscar_wins / helen.oscar_nominations))`?

The Constructor and Parameters

The Constructor and Parameters

Now imagine that you want to use the `Actor` class to instantiate an object for Helen Mirren and Tom Hanks.

```
class Actor:
    """Define the actor class"""
    def __init__(self):
        self.first_name = "Helen"
        self.last_name = "Mirren"
        self.birthday = "July 26"
        self.total_films = 80
        self.oscar_nominations = 4
        self.oscar_wins = 1

helen = Actor()
tom = Actor()

print(helen.first_name, helen.last_name)
print(tom.first_name, tom.last_name)
```

The `Actor` class only creates an object with information about Helen Mirren. You can make the `Actor` class more flexible by passing it a parameter for each of attributes in the constructor. Python still requires `self` as the first parameter. Parameters for the constructor method work just as they do for functions.

```
class Actor:
    """Define the actor class"""
    def __init__(self, first_name, last_name, birthday, total_films, o

        self.first_name = first_name
        self.last_name = last_name
        self.birthday = birthday
        self.total_films = total_films
        self.oscar_nominations = oscar_nominations
        self.oscar_wins = oscar_wins

helen = Actor("Helen", "Mirren", "July 26", 80, 4, 1)
tom = Actor("Tom", "Hanks", "July 9", 76, 5, 2)

print(helen.first_name, helen.last_name)
print(tom.first_name, tom.last_name)
```

challenge

What happens if you:

- Create an instace of the `Actor` class for Denzel Washington (December 28, 47 films, 8 nominations, 2 wins)?
- Add a print statement that say `print(help(Actor))`?

▼ **What does `help` do?**

The `help` function returns the documentation for objects, functions, keywords, etc. For a user-defined class, the `help` function returns the docstring (which is why a well-written docstring is useful) and any methods associated with the class. You should see the **`init`** method and its parameters.

Default Parameters

Like functions, classes can use default parameters for `__init__`. Most actors have not been nominated for an Oscar; even fewer win one. It is reasonable to assume that the "default" actor has 0 Oscar nominations and

0 wins. So these two parameters can default to 0 unless explicitly stated.

```
class Actor:
    """Define the actor class"""
    def __init__(self, first_name, last_name, birthday, total_films, o

        self.first_name = first_name
        self.last_name = last_name
        self.birthday = birthday
        self.total_films = total_films
        self.oscar_nominations = oscar_nominations
        self.oscar_wins = oscar_wins

helen = Actor("Helen", "Mirren", "July 26", 80, 4, 1)
dwayne = Actor("Dwayne", "Johnson", "July 9", 34)

print(f'{helen.first_name} won {helen.oscar_wins} oscar(s).')
print(f'{dwayne.first_name} won {dwayne.oscar_wins} oscar(s).')
```

challenge

What happens if you:

- Give the `birthday` parameter the default value of `"January 1"`?
- Give `total_films` parameter the default value of `10`?

▼ **Why did I get the error above?**

In Python, default parameters must come **at the end** of the parameter list. Both `oscar_nominations` and `oscar_wins` are at the end of the parameter list, so this does not cause a problem. When `birthday` is a default parameter, however, it is not at the end of the list. `total_films` comes after `birthday` and it requires a parameter. That is why you saw an error. When `birthday`, `total_films`, `oscar_nominations` and `oscar_wins` are all default parameters, the error message goes away.

Class Attributes

Class Attributes

Python also allows you to create a class attribute. The `__init__` method creates object attributes. These can be different from object to object. The objects `helen` and `dwayne` are completely different. However, a class attribute is the same for every object. The class attribute is defined after the docstring and before the `__init__` method.

```
class Actor:
    """Define the actor class"""
    union = "Screen Actors Guild"
    def __init__(self, first_name, last_name, birthday, total_films, o

        self.first_name = first_name
        self.last_name = last_name
        self.birthday = birthday
        self.total_films = total_films
        self.oscar_nominations = oscar_nominations
        self.oscar_wins = oscar_wins

helen = Actor("Helen", "Mirren", "July 26", 80, 4, 1)
dwayne = Actor("Dwayne", "Johnson", "July 9", 34)

print("{} {} is a member of the {}.".format(helen.first_name, helen.la

print("{} {} is a member of the {}.".format(dwayne.first_name, dwayne.
```

challenge

What happens if you:

- Add `helen.union = "Teamsters"` before the print statements and then run the program?
- Add the print statement `print(help(Actor))`?

Shallow & Deep Copies

Shallow Copies

The code below will instantiate `a` as an instance of the `ComicBookCharacter` class. Object `a` will be given a name, an age, and a type. Object `b` will be a copy of `a`. Finally, the `name` attribute of object `a` is changed. What do you expect to see when the `name` attributes of objects `a` and `b` are printed?

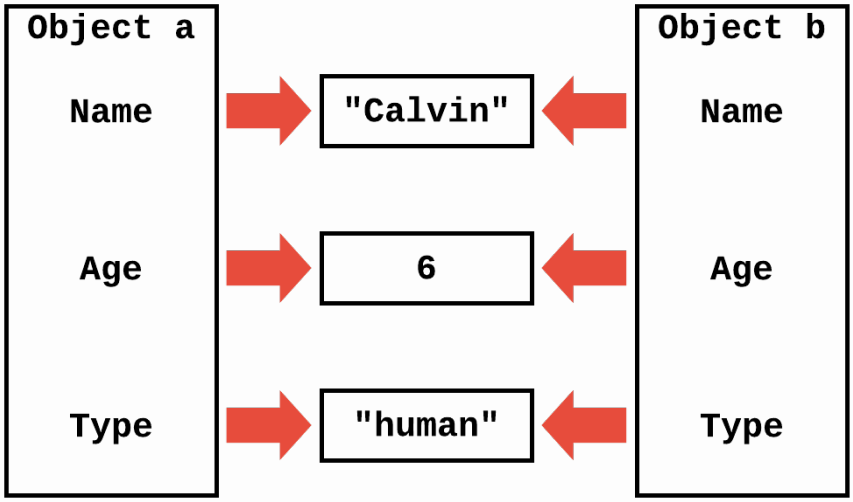
```
class ComicBookCharacter():
  pass

a = ComicBookCharacter()
a.name = "Calvin"
a.age = 6
a.type = "human"

b = a
a.name = "Hobbes"

print(a.name)
print(b.name)
```

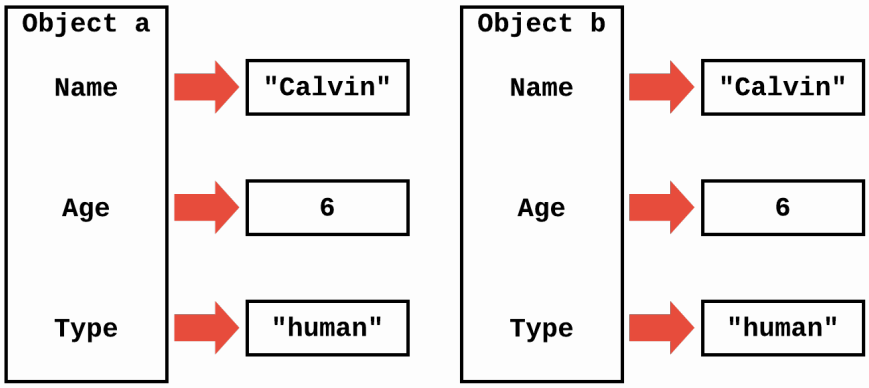
Both of the `name` attributes changed, even though the code only changed the `name` attribute of object `a`. This is because object `b` is a shallow copy of object `a`. Python makes a copy of object `a`, but object `b` shares the attributes with object `a`. That is why changing `name` for object `a` also affects the `name` attribute for object `b`.



Shallow Copy

Deep Copy

A deep copy is when Python makes a copy of object `a` and makes copies of each attribute. A deep copy keeps the attributes of one object independent of the other object. To create a deep copy, you need to import the `copy` module.



Deep Copy

```
import copy

class ComicBookCharacter():
  pass

a = ComicBookCharacter()
a.name = "Calvin"
a.age = 6
a.type = "human"

b = copy.deepcopy(a)
a.name = "Hobbes"

print(a.name)
print(b.name)
```