

סיכום שיעור שלישי:

1. אובייקטים :

- מהם?
- עולם אמיתי לעומת קוד.
- שימוש.
- מחלקה (class).
- יצירת מופע של מחלקה.
- Constructors

- Objects are key to understanding *object-oriented* paradigm.
- Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.
- Real-world objects share two characteristics: They all have **properties** and **actions**.
- Dogs have *properties* (name, color, breed, hungry) and **actions** (barking, fetching, wagging tail).
- Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.
- Later on each of objects will be represent as classes

<https://docs.python.org/3.4/tutorial/classes.html>

Class:

- A Class is a Group of Objects that has Common Properties.
- It is a Prototype , Template or Blueprint from which Objects are Created.
- All Classes Objects have the Same Properties but with different Values.
- All Objects Can Perform the Same Actions (Methods).

```
class Shark:  
    def bark():  
        print("whaf")
```

Creating an object:

- Once we created our class (**Shark**), we can create an object (**sammy**):

```
class Shark:
    def swim(self):
        print("The shark is swimming")

    def eat(self):
        print("The shark is eating")

def main():
    sammy = Shark()
    sammy.swim()
    sammy.eat()

if __name__ == "__main__":
    main()
```


Self:

- When objects are instantiated, the object itself is passed into the self parameter.
- Let's consider the example below.
- When we call **sammy.swim** python internally converts it for you as: **Shark.swim(sammy)**.
- Bottom line is the **Self** variable refers to the object itself.

```
class Shark:
    def swim(self):
        print("The shark is swimming")

def main():
    sammy = Shark()
    sammy.swim()

if __name__ == "__main__":
    main()
```



Object access:

In order to access object's methods there are two ways:

- Instantiate an object like in previous slide.
- Using a static method decoration above the method:

```
class Shark:
    @staticmethod
    def swim():
        print(132)

if __name__ == "__main__":
    Shark.swim()
```

- In order to access object's fields, we can instantiate a an object or declare a variable inside our class.
- In Python, variables declared inside the class definition, but not inside a method are class or static variables:

```
class Shark:
    name = "john"

if __name__ == "__main__":
    print(Shark.name)
```

:Packages and imports .2

- To use any module (.py files) in your code, you must first make it accessible by importing it.
- You can't use anything in Python before it is defined.
- Some things are built in, for example the basic types (like int, float, etc) can be used whenever you want.
- Most things you will want to do will need a little more than that.
- For example:

```
import datetime
print(datetime.datetime.now())
```

- Another way of using datetime is using the **from** with import:

```
from datetime import datetime
print(datetime.now())
```

- The difference between using only **Import** or using **import** and **from** is simple:
 - **Import** only – let us use a module (file) with no specific class (as we learned a module can contain a few classes), in which case we will need to specify which class we want to use each time.
 - **From** and Import- will specify the module and class we want to use.

<https://docs.python.org/3/tutorial/modules.html>

:Constructors .3

- A *constructor* is a special kind of method that Python calls when it instantiates an object using the definitions found in your class.
- Python relies on the constructor to perform tasks such as *initializing* (assigning values to) any instance variables that the object will need when it starts.
- Constructors can also verify that there are enough resources for the object and perform any other start-up task you can think of.
- The name of a constructor is always the same, **__init__()**.
- The constructor can accept arguments when necessary to create the object.
- When you create a class without a constructor, Python automatically creates a default constructor for you that doesn't do anything.
- Every class must have a constructor, even if it simply relies on the default constructor.

- This is how we can instantiate sammy with an age:
 1. Create a constructor (`__init__` function) which accept a variable / variables.
 2. Instantiating an object with the desired value (3).

```
class Shark:
    def __init__(self, age):
        self.age = age
        print(age)

def main():
    stevie = Shark(3)

if __name__ == "__main__":
    main()
```

- Inheritance is a powerful feature in object oriented programming.
- It refers to defining a new class with little or no modification to an existing class.
- The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.
- Lets all agree that any car has model and color, regardless to anything.
- Of course we can work in the non-oop way, which is creating each class with all parameters and not using inheritance.
- We will just find ourselves multiplying a lot of code!
- There are 2 options to use:
 - a. The first option will be, using the parent class `_init_` method and passing the required parameters:

```
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color

class ElectricCar(Car):
    def __init__(self, model, color, battery_type):
        Car.__init__(self, model, color)
        self.battery_type = battery_type

def main():
    regular_volvo = Car("volvo", "white")
    electric_volvo = ElectricCar("electronic_volvo", "red", "lithium")

if __name__ == "__main__":
    main()
```

- b. The second option will be, using the **super** keyword along with `_init_` method and passing the required parameters:

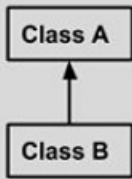
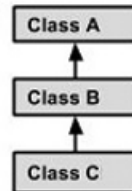
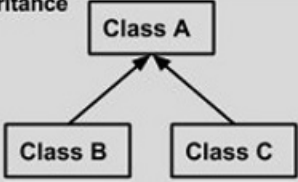
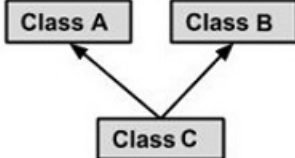
```
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color

class ElectricCar(Car):
    def __init__(self, model, color, battery_type):
        super(ElectricCar, self).__init__(model, color)
        self.battery_type = battery_type

def main():
    regular_volvo = Car("volvo", "white")
    electric_volvo = ElectricCar("electronic_volvo", "red", "lithium")

if __name__ == "__main__":
    main()
```

Inheritance types:

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> Class Car: ... Class Volvo(Car): ... </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> Class Car: ... Class Volvo(Car): ... Class VolvoS80(Volvo): ... </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> Class Car: ... Class Volvo(Car): ... Class Fiat(Car): ... </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> Class Car: ... Class Vehicle: ... Class Volvo(Car, Vehicle): ... </pre>

:Data structure .5

<https://docs.python.org/3/tutorial/datastructures.html>

Array:

- Python provides a data structure, the **array**, which stores a collection of elements of the same type.
- Arrays are used to store a few elements of the same type so we can use them later.
- Look at array as a stack that holds a few elements.
- Arrays always start from 0.
- To define an array, we need to add 3 things:
 - Import array
 - Initialize the array with the desired type (look partial at table)
 - Adding elements

```

import array as arr
a = arr.array("i", [3, 6, 9])
        
```

Type code	Python type
c	char
i	int
f	float

Array usage:

- Adding an element:

```
a = arr.array('i',[3,6,9])  
a.append(111)
```

- Removing an element:

```
a = arr.array('i',[3,6,9])  
a.pop(0)
```

- Modify existing value in a specific index

```
a = arr.array('i',[3,6,9])  
a[0] = 5
```

- Get a value from a specific index

```
a = arr.array('i',[3,6,9])  
print(a[0])
```

- Adding an element to a specific index

```
a = arr.array('i',[3,6,9])  
a.insert(1,7)
```

- Get the array size (number of elements) using len:

```
a = arr.array('i',[3,6,9])  
print(len(a))
```

- Iterating the array and getting all elements

- Without index:

```
a = arr.array('i',[3,6,9])  
for temp_num in a:  
    print(temp_num)
```

- Without index

```
a = arr.array('i',[3,6,9])  
for i in range(len(a)):  
    print(a[i])
```


List:

- Lists in Python are used to store collection of heterogeneous items.
- These are mutable, which means that you can change their content without changing their identity.
- You can recognize lists by their square brackets [and] that hold elements, separated by a comma:

```
my_list = [5,"a",True]
```

- List shares the same functions as array!

Tuple:

- Tuple is a data structure very similar to the **list** data structure.
- The main difference being that tuple manipulation are faster than list because tuples are immutable, which means once defined you cannot delete, add or edit any values inside it.
- The simple usage can be, when we already know what data is stored, and we don't want it to change (for example: seasons)
- Another usage is in situations where you want to pass the data to someone else but you do not want them to manipulate data in your collection.
- Tuple can be written with/without brackets:

```
x_tuple = 1, 2, 3, 4, 5  
y_tuple = ('a','b','c','d')
```

Dictionary:

- A dictionary is a sequence of items where each item is a pair made of a key and a value.
- Dictionaries are not sorted, so you can access to the list of keys or values independently.
- Dictionaries are surrounded by curly brackets:

```
my_dictionary = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
```

- Dictionaries come with many build in options, such as changing a specific value:

```
my_dictionary['A'] = 5
```

- Getting all dictionary keys/values:

```
Print(my_dictionary.keys()) → Print(my_dictionary.values())
```

- Deleting a pair

```
del(my_dictionary['A'])
```

- A full list of data structures functions can be found here:

<https://docs.python.org/3.3/tutorial/datastructures.html>

תזכורת ממה שעשינו בכיתה

```
# classes and objects
class Shark:
    def swim(self):
        print("The shark is swimming")

    def eat(self):
        print("The shark is eating")

def main():
    sammy = Shark()
    sammy.swim()
    sammy.eat()

if __name__ == "__main__":
    main()

# static example
class StaticShark:
    @staticmethod
    def swim():
        print(132)

if __name__ == "__main__":
    StaticShark.swim()

# constructors
class ConstructedShark:
    def __init__(self, age):
        self.age = age
        print(age)

def main():
    stevie = ConstructedShark(3)

if __name__ == "__main__":
    main()

# inheritance
class Car:
    def __init__(self, model, color):
        self.model = model
        self.color = color

class ElectricCar(Car):
    def __init__(self, model, color, battery_type):
        super(ElectricCar, self).__init__(model, color)
        self.battery_type = battery_type

def main():
    regular_volvo = Car("volvo", "white")
    electric_volvo = ElectricCar("electronic_volvo", "red", "lithium")
```

```
# data structures - array
import array as arr
a = arr.array("i", [3, 6, 9])
a[0] = 5
a.append(7)
print(a[0])
a.pop(1)
a.insert(0, 1)

for temp_num in a:
    print(temp_num)

# list
my_list = [5, "a", True]

#tuple
x_tuple = 1, 2, 3, 4, 5

#dictionary
my_dictionary = {'A': 1, 'B': 2, 'C': 3, 'D': 4}
print(my_dictionary['A'])
```