|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| C:\Users\Admin\Desktop\download.jpg | USMAN INSTITUTE OF TECHNOLOGY | | | | | |  |
|  |  |  |  |  |  |  |  |
|  | Department of Computer Science  CS121 Object Oriented Programming | | | | | |  |
|  |  | Lab # 06  Objects and Classes in Python | | | |  |  |
|  | Objective:  This experiment introduces the students to the concept of Objects and Classes using Python as a Programming Language. A brief introduction to the Class Diagram of the Unified Modeling Language is also covered | | | | | |  |
|  | **Name of Student:**  **Roll No: Sec.**  **Date of Experiment:** | | | | | |  |
|  | **Marks Obtained/Remarks:**  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  **Signature:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | | | |  |

**Lab 06: Objects and Classes in Python**

# Introducing Object Oriented

Everyone knows what an object is—a tangible thing that we can sense, feel, and manipulate. The earliest objects we interact with are typically baby toys. Wooden blocks, plastic shapes, and oversized puzzle pieces are common first objects. Babies learn quickly that certain objects do certain things: bells ring, buttons press, and levers pull. The definition of an object in software development is not terribly different. Software objects are not typically tangible things that you can pick up, sense, or feel, but they are models of something that can do certain things and have certain things done to them. Formally, an object is a collection of data and associated behaviors.

So, knowing what an object is, what does it mean to be object-oriented? Oriented simply means

directed toward. So object-oriented means functionally directed towards modeling objects. This is one of the many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior.

If you've read any hype, you've probably come across the terms object-oriented analysis, object oriented design, object-oriented analysis and design, and object oriented programming. These are all highly related concepts under the general object-oriented umbrella.

In fact, analysis, design, and programming are all stages of software development. Calling them

object-oriented simply specifies what style of software development is being pursued.

**Object-oriented analysis (OOA)** is the process of looking at a problem, system, or task (that somebody wants to turn into an application) and identifying the objects and interactions between those objects. The analysis stage is all about what needs to be done. The output of the analysis stage is a set of requirements. If we were to complete the analysis stage in one step, we would have turned a task, such as, I need a website, into a set of requirements.

For example: Website visitors need to be able to (italic represents actions, bold represents objects):

* *review* our **history**
* *apply* for **jobs**
* *browse, compare*, and *order* **products**

In some ways, analysis is a misnomer. The baby we discussed earlier doesn't analyze the blocks and puzzle pieces. Rather, it will explore its environment, manipulate shapes, and see where they might fit. A better turn of phrase might be object-oriented exploration. In software development, the initial stages of analysis include interviewing customers, studying their processes, and eliminating possibilities. Object-oriented design (OOD) is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify which objects can activate specific behaviors on other objects. The design stage is all about how things should be done. The output of the design stage is an implementation specification. If we were to complete the design stage in a single step, we would have turned the requirements defined during object-oriented analysis into a set of classes and interfaces that could be implemented in (ideally) any object-oriented programming language.

**Object-oriented programming (OOP)** is the process of converting this perfectly defined design into a working program that does exactly what the CEO originally requested.

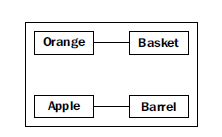
# Objects and Classes

So, an object is a collection of data with associated behaviors. How do we differentiate between types of objects? Apples and oranges are both objects, but it is a common adage that they cannot be compared. Apples and oranges aren't modeled very often in computer programming, but let's pretend we're doing an inventory application for a fruit farm. To facilitate the example, we can assume that apples go in barrels and oranges go in baskets.

Now, we have four kinds of objects: apples, oranges, baskets, and barrels. In object-oriented modeling, the term used for *kind of object* is **class**. So, in technical terms, we now have four classes of objects.

What's the difference between an object and a class? Classes describe objects. They are like blueprints for creating an object. You might have three oranges sitting on the table in front of you. Each orange is a distinct object, but all three have the attributes and behaviors associated with one class: the general class of oranges

The relationship between the four classes of objects in our inventory system can be described using a Unified Modeling Language (invariably referred to as UML, because three letter acronyms never go out of style) class diagram. Here is our first class diagram.



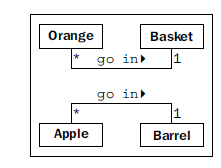
This diagram shows that an Orange is somehow associated with a Basket and that an Apple is also somehow associated with a Barrel. Association is the most basic way for two classes to be related

# Introduction to UML

UML is very popular among managers, and occasionally disparaged by programmers. The syntax of a UML diagram is generally pretty obvious; you don't have to read a tutorial to (mostly) understand what is going on when you see one. UML is also fairly easy to draw, and quite intuitive. After all, many people, when describing classes and their relationships, will naturally draw boxes with lines between them. Having a standard based on these intuitive diagrams makes it easy for programmers to communicate with designers, managers, and each other.

Our initial diagram, while correct, does not remind us that apples go in barrels or how many barrels a single apple can go in. It only tells us that apples are somehow associated with barrels. The association between classes is often obvious and needs no further explanation, but we have the option to add further clarification as needed.

The beauty of UML is that most things are optional. We only need to specify as much information in a diagram as makes sense for the current situation. In a quick whiteboard session, we might just quickly draw lines between boxes. In a formal document, we might go into more detail. In the case of apples and barrels, we can be fairly confident that the association is, many apples go in one barrel, but just to make sure nobody confuses it with, one apple spoils one barrel, we can enhance the diagram as shown.



This diagram tells us that oranges go in baskets with a little arrow showing what goes in what. It also tells us the number of that object that can be used in the association on both sides of the relationship. One Basket can hold many (represented by a \*) Orange objects. Any one Orange can go in exactly one Basket. This number is referred to as the multiplicity of the object. You may also hear it described as the cardinality. These are actually slightly distinct terms. Cardinality refers to the actual number of items in the set, whereas multiplicity specifies how small or how large this number could be.

# Classes And Objects in Python

All class definitions start with the class keyword, which is followed by the name of the class and a colon Any code that is indented below the class definition is considered part of the class’s body.

*Example 1*

* **class** Dog:
* pass

Python class names are written in Capitalized Words notation by convention

**Defining Attributes and Initializer Method**

A special method to initialize a new object’s state when it is created. It can perform any action, but designed to perform initialization actions. It is used for creating an object’s data fields with initial values

Let’s define some properties that all Dog objects should have A number of properties that can be chosen from include

* **name, age**, coat color, and breed

The properties that all Dog objects must have are defined in a method called .\_\_init\_\_()

Every time a new Dog object is created, .\_\_init\_\_() sets the initial state of the object by assigning the values of the object’s properties. .\_\_init\_\_() initializes each new instance of the class

.\_\_init\_\_() can any number of parameters. However the first parameter will always be a variable called **self**. When a new class instance is created, the instance is automatically passed to the self parameter in .\_\_init\_\_() so that new attributes can be defined on the object

*Example 2*

* **class** Dog:
* def \_\_init\_\_(self, name, age):
* self.name = name
* self.age = age

**Instance versus Class Attributes**

Attributes created in .\_\_init\_\_() are called instance attributes. An instance attribute’s value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

Class attributes are attributes that have the same value for all class instances. We can define a class attribute by assigning a value to a variable name outside of .\_\_init\_\_().

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

*Example 3*

* **class** Dog:
* # Class attribute
* species = "Canis familiaris"
* def \_\_init\_\_(self, name, age):
* self.name = name
* self.age = age

**Defining Methods**

* **class** ClassName:
* initializer
* **methods**

All methods, including initializer, have the first parameter ***self. Self*** refers to the object that invokes / calls the method. Any name can be specified for this parameter; convention is **self** is used.

**Object Instantiation**

Creating a new object from a class is called instantiating an object. We can instantiate a new Dog object by typing the name of the class, followed by opening and closing parentheses

To instantiate objects of the defined Dog class, we need to provide values for the name and age. To pass arguments to the name and age parameters, put values into the parentheses after the class name.

*Example 4*

* **class** Dog:
* # Class attribute
* species = "Canis familiaris"
* def \_\_init\_\_(self, name, age):
* self.name = name
* self.age = age
* buddy = Dog("Buddy", 9)
* miles = Dog("Miles", 4)

This creates two new Dog instances—one for a nine-year-old dog named Buddy and one for a four-year-old dog named Miles.

**Dot Notation**

Object’s data fields and methods are accessed by using dot operator (.) It is also known as ***Object Member Access Operator.***

* objectRefVar.dataField
* objectRefVar.method(arguments)

After we create the Dog instances, we can access their instance attributes using dot notation

* print(buddy.name)
* 'Buddy'
* Print(buddy.age)
* 9

# Student Exercise

TV Sets

Each TV is an object with states (that is, current channel, current volume level, and power on or off are its properties that are represented by data fields) and behaviors (change channels, adjust volume and turn on/off are the actions each TV object implements with methods).

UML diagram for the TV class is shown below

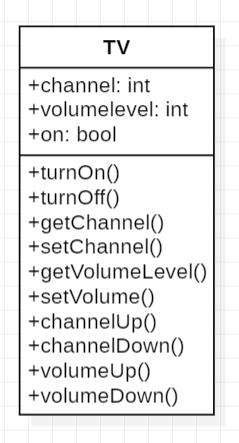
|  |
| --- |
| **TV** |
| channel: int  volumeLevel: int  on: bool |
| TV()  turnOn(): None  turnoff(): None  getChannel(): int  setChannel(channel: int): None  getVolume(): int  setVolume(volumeLevel: int): None  channelUp(): None  channelDown(): None  volumeUp(): None  volumeDown(): None |

|  |
| --- |
|  |
| *The current channel (1 to 120) for this TV*  *The current volume (1 to 7) of this TV*  *Indicates whether this TV is on/off* |
| *Constructs a default TV object*  *Turns on this TV*  *Turns off this TV*  *Returns the channel for this TV*  *Sets a new channel for this TV*  *Gets the volume level for this TV*  *Sets a new volume level for this TV*  *Increase the channel by 1*  *Decrease the channel by 1*  *Increase the volume by 1*  *Decrease the channel by 1* |

Exercise 1

Write Python code for defining the TV class

**UML Diagram:**

****

**Code:**

class TV:

def \_\_init\_\_(self):

self.channel = 0

self.volumeLevel = 0

self.on = False

def turnOn(self):

self.on = True

def turnOff(self):

self.on = False

def getChannel(self):

return self.channel

def setChannel(self, channel):

if self.on and 0 <= channel <= 120:

self.channel = channel

def getVolumeLevel(self):

return self.volumeLevel

def setVolume(self, volumeLevel):

if self.on and 0 <= volumeLevel <= 7:

self.volumeLevel = volumeLevel

def channelUp(self):

if self.on and self.channel < 120:

self.channel += 1

def channelDown(self):

if self.on and self.channel > 0:

self.channel -= 1

def volumeUp(self):

if self.on and self.volumeLevel < 7:

self.volumeLevel += 1

def volumeDown(self):

if self.on and self.volumeLevel > 0:

self.volumeLevel -= 1

Exercise 2

Implement a program to create two TV objects. The program turns on the first TV, set channel to 30 and set volume to level 3. The program then turns on the second TV, twice increasing its channel by 1, and increase its volume by 1. The program finally displays the state of both the objects.

**Code:**

#Exercise 2

tv1 = TV()

tv1.turnOn()

tv1.setChannel(30)

tv1.setVolume(3)

print("tv1's channel is", tv1.getChannel(), "and volume level is", tv1.getVolumeLevel())

tv2 = TV()

tv2.turnOn()

tv2.channelUp()

tv2.channelUp()

tv2.volumeUp()

print("tv2's channel is", tv2.getChannel(), "and volume level is", tv2.getVolumeLevel())

**Output:**

