**CHAPTER 1**

**CLASSES AND OBJECTS**

## Programmer-defined types

* + *programmer-defined types are created using the `class` keyword. By defining a class, you can create your own custom data types with properties (attributes) and behaviors (methods) specific to your application.*

*class Person:*

*def \_init\_(self, name, age):*

*self.name = name*

*self.age = age*

*def say\_hello(self):*

*print("Hello, my name is", self.name)*

*def get\_age(self):*

*return self.age*

*# Create an instance of the Person class*

*person1 = Person("Alice", 25)*

*# Access the attributes and call the methods of the person1 object*

*print(person1.name)*

*print(person1.get\_age())*

*person1.say\_hello()*

**Attributes**

*an attribute is a variable that belongs to an object. It holds data specific to that object and can be accessed and modified using dot notation (`object.attribute`).*

*Here's a short program that demonstrates the use of attributes in Python:*

*python*

*class Person:*

*def \_init\_(self, name, age):*

*self.name = name*

*self.age = age*

*# Create an instance of the Person class*

*person1 = Person("Alice", 25)*

*# Access and modify attributes*

*print(person1.name) # Output: Alice*

*person1.age = 30*

*print(person1.age)  # Output: 30*

## Rectangles

* + Here is the class definition:

class Rectangle: """Represents a rectangle.

attributes: width, height, corner."""

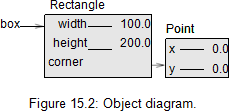
* The doc string lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.
* To represent a rectangle, you have to instantiate a Rectangle object and assign values to theattributes:

box = Rectangle() box.width = 100.0

box.height = 200.0 box.corner = Point() box.corner.x = 0.0

box.corner.y = 0.0

* + The expression box.corner.x means, “Go to the object box refers to and select the attributenamed corner; then go to that object and select the attribute named x.”
  + Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.



## Instances as return values

* + Functions can return instances. For example, find\_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle

def find\_center(rect):

p = Point()

p.x = rect.corner.x + rect.width/2

p.y = rect.corner.y + rect.height/2return p

* + Here is an example that passes box as an argument and assigns the resulting Point to center.

center = find\_center(box) print\_point(center)

(50, 100)

## Objects are mutable

* + You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

box.width = box.width + 50 box.height = box.height + 100

* + You can also write functions that modify objects.
  + For example, grow\_rectangle takes a Rectangle object and two numbers, dwidth and dheight,and adds the numbers to the width and height of the rectangle:

def grow\_rectangle(rect, dwidth, dheight): rect.width += dwidth

rect.height += dheight

* + Here is an example that demonstrates the effect:

box.width, box.height (150.0, 300.0)

grow\_rectangle(box, 50, 100) box.width, box.height (200.0, 400.0)

* + Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

## Copying

* + Copying an object is often an alternative to aliasing. The copy module contains a function calledcopy that can duplicate any object:

*A shallow copy creates a new object that contains the same references as the original object. This means that any changes made to the new object will also be reflected in the original object.*

*A deep copy creates a new object that contains the same values as the original object, but not the same references. This means that changes made to the new object will not be reflected in the original object.*

*Here is an example of a shallow copy:*

*>>> a = [1, 2, 3]*

*>>> b = a[:]*

*>>> b[0] = 4*

*>>> a*

*[4, 2, 3]*

*>>> b*

*[4, 2, 3]*

*Here is an example of a deep copy:*

*>>> import copy*

*>>> a = [1, 2, 3]*

*>>> b = copy.deepcopy(a)*

*>>> b[0] = 4*

*>>> a*

*[1, 2, 3]*

*>>> b*

*[4, 2, 3]*

*Shallow copy is usually used when you want to create a copy of an object that you do not want to modify. For example, you might use a shallow copy to create a backup of a file.*

*Deep copy is usually used when you want to create a copy of an object that you want to modify without affecting the original object. For example, you might use a deep copy to create a new list that you can then sort without affecting the original list.*

# CHAPTER 02

**CLASSES AND FUNCTIONS**

* + As another example of a programmer-defined type, we’ll define a class called Time that records the time of day. The class definition looks like this:

class Time:

"""Represents the time of day. attributes: hour, minute, second """

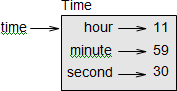
* We can create a new Time object and assign attributes for hours, minutes, and seconds:

time = Time( ) time.hour = 11

time.minute = 59

time.second = 30

* The state diagram for the Time object looks like Figure below.



***What is classes and objects give example with syntax?***

**Class:** a class is a userdefineddata types consists of data members and member functions associated with it

It also consists of functions

**Syntax:**

class classname:

block of code

**Example:**

class sample:

a=int(input(“enter the value of a”))

b=a

print(“the value of b is” ,b)

**Object:** the object is an instance of a class and it is also a real world entity

**Syntax:**

class classname:

block of code

object=classname()

**Example:**

class sample:

a=int(input(“enter the value of a”))

b=a

print(“the value of b is” ,b)

s=sample()

## Pure functions

* + In the next few sections, we’ll write two functions that add time values.
  + They demonstrate two kinds of functions: pure functions and modifiers.
  + They also demonstrate a development plan I’ll call prototype and patch, which is a way of tackling acomplex problem by starting with a simple prototype and incrementally dealing with the complications.

def add\_time(t1, t2): sum = Time()

sum.hour = t1.hour + t2.hour

sum.minute = t1.minute + t2.minute sum.second = t1.second + t2.second return sum

* + Here is a simple prototype of add\_time:
  + The function creates a new Time object, initializes its attributes, and returns a reference to the new object
  + ***This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value***

***(Or)***

* + *A function is called*[*pure function*](http://en.wikipedia.org/wiki/Pure_function)*if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something. The only result of calling a pure function is the return value. Examples of pure functions are strlen(), pow(), sqrt() etc*
  + To test this function, let us create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one hour 35 minutes.
  + add\_time figures out when the movie will be done

>>> start = Time()

>>> start.hour = 9

>>> start.minute = 45

>>> start.second = 0

>>> duration = Time()

>>> duration.hour = 1

>>> duration.minute = 35

>>> duration.second = 0

>>> done = add\_time(start, duration)

>>> print\_time(done)

10:80:00

1. The result, 10:80:00 might not be what you were hoping for.
2. The problem is that this function does not deal with cases where the number of seconds orminutes adds up to more than sixty.
   * When that happens, we have to “carry” the extra seconds into the minute column or the extra minutes into the hour column.
   * Here’s an improved version:

def add\_time(t1, t2):

sum = Time()

sum.hour = t1.hour + t2.hour sum.minute = t1.minute + t2.minute sum.second = t1.second + t2.second

if sum.second >= 60: sum.second -= 60

sum.minute += 1

if sum.minute >= 60: sum.minute -= 60

sum.hour += 1

return sum

***Impure Functions:****An impure function is a function that has side effects or does not always return the same output when given the same input. Side effects can include modifying a global variable, changing the state of an object, or making a network request.*

*Impure functions are harder to reason about and test because they have hidden dependencies and can produce different outputs for the same input.*

**Here are some different examples of impure functions:**

**Example 1:** Functions that modify a global variable

* Javascript

|  |
| --- |
| let oldValue = 7;    **function** add(newValue) {  **return** oldValue += newValue;  }    console.log(add(5)); |

**Output**

12

## Modifiers

## *In programming, modifiers are keywords or syntax elements that change the behavior or characteristics of a variable, function, or class. Modifiers are used to define the access level, storage location, or other properties of the entities in a program.*

## *Some common modifiers in programming languages include:*

## *1. Access Modifiers: These modifiers control the visibility and accessibility of variables, functions, or classes. Examples include `public`, `private`, and `protected` in languages like Java and C++.*

## *2. Static Modifier: The static modifier is used to declare variables and methods that belong to a class, rather than an instance of the class. Static variables are shared among all instances of a class, and static methods can be called without creating an object of the class.*

## *3. Final Modifier: The final modifier is used to indicate that a variable, method, or class cannot be modified or overridden. In Java, for example, a final variable cannot be reassigned, a final method cannot be overridden, and a final class cannot be subclassed.*

## *4. Abstract Modifier: The abstract modifier is used to declare abstract classes and methods. An abstract class cannot be instantiated and serves as a base for subclasses. Abstract methods are declared without an implementation and must be implemented by concrete subclasses.*

## *5. Volatile Modifier: The volatile modifier is used to indicate that a variable may be modified by multiple threads, and its value should always be read from the main memory rather than a cache.*

## *These are just a few examples of modifiers commonly found in programming languages. Different languages may have their own set of modifiers with specific functionalities and use cases.*

def increment(time, seconds): time.second += seconds

if time.second >= 60: time.second -= 60

time.minute += 1

if time.minute >= 60: time.minute -= 60

time.hour += 1

**PROTOTYPE AND PATCH**

*The "prototype and patch" is a program development concept that involves rapidly prototyping a solution and then refining it through a series of patches or incremental changes. This approach allows for quick iterations and improvements based on feedback and testing. Here's an explanation with a suitable example:*

*1. Prototype:*

*- In the prototyping phase, a basic version or a proof-of-concept of the program is developed quickly.*

*- The focus is on implementing the core functionality and testing it for feasibility and effectiveness.*

*- The prototype may not have all the desired features or a polished user interface.*

*Example:*

*Let's say you are developing a weather application. In the prototyping phase, you might create a basic version that fetches weather data for a single location and displays it in a simple text format. The prototype may not have additional features like multiple locations, graphical representation, or user customization options. The goal is to quickly validate the basic functionality of fetching and displaying weather data.*

*2. Patch:*

*- After the prototype is developed, it is tested and evaluated for any issues or limitations.*

*- Based on the feedback and requirements, incremental changes or "patches" are made to improve the program.*

*- Patches can include bug fixes, adding new features, enhancing the user interface, or optimizing the code.*

*Example:*

*In the weather application example, after the prototype is tested, you receive feedback from users requesting additional features like a graphical representation of temperature trends and the ability to add multiple locations. In the patching phase, you would make incremental changes to implement these features. This may involve updating the code to fetch data for multiple locations, designing a visual representation of temperature trends using charts or graphs, and enhancing the user interface to allow users to customize their preferred settings.*

*The patching phase continues iteratively as the program evolves based on user feedback, bug reports, and new requirements. Each patch improves the program's functionality, usability, and performance.*

*The "prototype and patch" approach allows for flexibility and quick iterations in program development. It helps in validating ideas, getting early user feedback, and gradually refining the program to meet the desired goals*.

## CHAPTER 03

**CLASSES AND METHODS**

# Object-Oriented Features

* + - Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:
* Programs include class and method definitions.
* Most of the computation is expressed in terms of operations on objects.
* Objects often represent things in the real world, and methods often correspond to the ways things in thereal world interact.
  + - A method is a function that is associated with a particular class.
    - Methods are semantically the same as functions, but there are two syntactic differences:
* Methods are defined inside a class definition in order to make the relationship between the class and themethod explicit.
* The syntax for invoking a method is different from the syntax for calling a function

# Printing Objects

* + We already defined- a class named and also wrote a function named print\_time:

def print\_time(time):

print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))

class Time:

"""Represents the time of day."""

* + To call this function, we have to pass a Time object as an argument:

>>> start = Time()

>>> start.hour = 9

>>> start.minute = 45

>>> start.second = 00

>>>> print\_time(start) 09:45:00

* To make print\_time a method, all we have to do is move the function definition inside the class definition.Notice the change in indentation.

class Time:

def print\_time(time):

print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))

* + Now there are two ways to call print\_time. The first (and less common) way is to use functionsyntax:

>>>Time.print\_time(start) 09:45:00

* + In this use of dot notation, Time is the name of the class, and
  + print\_time is the name of themethod. start is passed parameter.
  + The second (and more concise) way is to use method syntax:

>>> start.print\_time() 09:45:00

* + In this use of dot notation, print\_time is the name of the method (again), and start is the object the methodis invoked on, which is called the subject.
    - Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is whatthe method is about.
    - Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.
    - By convention, the first parameter of a method is called self, so it would be more common to writeprint\_time like this:

class Time:

def print\_time(self):

print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))

* The reason for this convention is an implicit metaphor:
* The syntax for a function call, print\_time(start), suggests that the function is the active agent. It sayssomething like, “Hey print\_time! Here’s an object for you to print.”
* In object-oriented programming, the objects are the active agents. method invocation like start.print\_time() says “Hey start! Please print yourself.”

# 3, Another Example

* Here’s a version of increment rewritten as a method:

# inside class Time:

def increment(self, seconds): seconds += self.time\_to\_int() return int\_to\_time(seconds)

* This version assumes that time\_to\_int is written as a method. Also, note that it is a pure function, not amodifier.
* Here’s how you would invoke increment:

>>> start.print\_time()

09:45:00

>>> end = start.increment(1337)

>>> end.print\_time() 10:07:17

* The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to thesecond parameter, seconds.
* This mechanism can be confusing, especially if you make an error. For example, if you invokeincrement with two arguments, you get:

>>> end = start.increment(1337, 460)

TypeError: increment() takes 2 positional arguments but 3 were given

* The error message is initially confusing, because there are only two argument in parentheses. But thesubject is also considered an argument, so all together that’s three.
* By the way, a positional argument is an argument that doesn’t have a parameter name; that is, it is not akeyword argument. In this function call:

sketch(parrot, cage, dead=True)

* parrot and cage are positional, and dead is a keyword argument.

# The init Method

* + The init method (short for “initialization”) is a special method that gets invoked when an object isinstantiated.
  + ***\_\_init\_\_ is a special method known as the constructor. It is automatically called when a new instance (object) of a class is created. The \_\_init\_\_ method allows you to initialize the attributes (variables) of an object.***
  + Its full name is init (two underscore characters, followed by init, and then two more underscores).
  + An init method for the Time class might look like this:

# inside class Time: def \_init (self, hour=0,

minute=0, second=0): self.hour = hour self.minute = minute self.second = second

* It is common for the parameters of init to have the same names as the attributes.
  + The statement

self.hour = hour

* + stores the value of the parameter hour as an attribute of self.
  + The parameters are optional, so if you call Time with no arguments, you get the default values:

>>> time = Time()

>>> time.print\_time()

00:00:00

* If we provide one argument, it overrides hour:

>>> time = Time (9)

>>> time.print\_time()

09:00:00

* If we provide two arguments, they override hour and minute.

>>> time = Time(9, 45)

>>> time.print\_time() 09:45:00

* And if we provide three arguments, they override all three default values

# The \_str\_ Method

# inside class Time:

def str (self):

return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

* str is a special method, like init , that is supposed to return a string representa- tion of anobject.
* ***\_\_str\_\_ is a special method, like \_\_init\_\_ , that is supposed to return a string representation of an object.***
* For example, here is a str method for Time object
* When you print an object, Python invokes the str method:

>>> time = Time(9, 45)

>>> print(time) 09:45:00

# Operator Overloading

* + By defining other special methods, you can specify the behavior of operators on programmer-defined types.
  + For example, if we define a method named add for the Time class, you can use the + operator on Timeobjects.
  + Here is what the definition might look like:

def \_add\_(self,other): seconds=self.time\_to\_int()+other.time\_to\_int()

return int\_to\_time(seconds)

* And here is how we could use it:

>>> start = Time(9, 45)

>>> duration = Time(1, 35)

>>> print(start + duration) 11:20:00

* When you apply the + operator to Time objects, Python invokes add .
* When you print the result, Python invokes str . So there is a lot happening behind the scenes!
* Changing the behavior of an operator so that it works with programmer-defined types is called operatoroverloading.
* ***Operator Overloading****means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class.*
* For every operator in Python there is a corresponding special method, like add .

# Type-Based Dispatch

* + The following is the version of \_add\_ that checks the type of other and invokes either add\_time orincrement:

def add (self,other):

if isintance(other, Time): return self.add\_time(other)

else:

return self.increment(other)

def add\_time(self, other):

seconds = self.time\_to\_int() + other.time\_to\_int() return int\_to\_time(seconds)

def increment(self, seconds): seconds += self.time\_to\_int() return int\_to\_time(seconds)

* + The built-in function isinstance takes a value and a class object, and returns True if the value is an instanceof the class.
  + If other is a Time object, add invokes add\_time. Otherwise it assumes that the parameter is a numberand invokes increment.

***Type dispatch, or Multiple dispatch, allows you to change the way a function behaves based upon the input types it recevies.***

* + This operation is called a type-based dispatch because it dispatches the computation to different methods based on the type of the arguments.
  + Here are examples that use the + operator with different types:

>>> start = Time(9, 45)

>>> duration = Time(1, 35)

>>> print(start + duration) 11:20:00

>>> print(start + 1337)

10:07:17

* + Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

>>> print(1337 + start)

TypeError: unsupported operand type(s) for +: 'int' and 'instance'

* + The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add aTime object, and it doesn’t know how.
  + But there is a clever solution for this problem: the special method radd , which stands for “right-sideadd”.
  + This method is invoked when a Time object appears on the right side of the + operator. Here’s thedefinition:

# inside class Time:

def radd (self, other): return self. add (other)

>>> print(1337 + start) 10:07:17

# Polymorphism

* + - Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.
    - Many of the functions we wrote for strings also work for other sequence types. For example, we usedhistogram to count the number of times each letter appears in a word.

def histogram(s):

d = dict()

for c in s:

if c not in d: d[c] = 1

else:

d[c] = d[c]+1 return d

* + This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, sothey can be used as keys in d:

>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']

>>> histogram(t)

{'bacon': 1, 'egg': 1, 'spam': 4}

* + Functions that work with several types are called polymorphic. Polymorphism can facilitate code reuse.
  + For example, the built-in function sum, which adds the elements of a sequence, works as long as theelements of the sequence support addition.

>>> t1 = Time(7, 43)

>>> t2 = Time(7, 31)

>>> t3 = Time(7, 37)

>>> total = sum(t1, t2, t3)

>>> print(total) 23:01:00

* + In general, if all of the operations inside a function work with a given type, the function works with thattype.
  + The best kind of polymorphism is the unintentional kind, where you discover that a func- tion you alreadywrote can be applied to a type you never planned for.

***Questions from previous semesters***

|  |
| --- |
| ***Define the terms with example: (i) class (ii) objects (iii) instance variables***  *i. Class :*  *Python is an object-oriented programming language, and class is a basis for any object oriented programming language.*  *Class is a user-defined data type which consists of data members and member functions together into single entity*  *. Class is just a prototype (or a logical entity/blue print) which will not consume any memory.*  *Example:*  *class Time:*  *‘‘‘Represents a Time of a day.*  *Attributes: hour, minute, second’’’*  *ii. Object:*  *An object is an instance of a class and it has physical existence.*  *t1 = Time( )*  *t2 = Time( )*  *iii. Instance variable*  *A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods). Python instance variables are owned by an instance of a class. The value of an instance variable can be different depending on the instance with which the variable is associated.*  *t1 . hour = 10*  *t1 . minute = 30*  *t1 . second = 30*  *t2 . hour = 2*  *t2 . minute = 10*  *t2 . second = 10* |
| ***Create a Time class with hour, min and sec as attributes. Demonstrate how two Time objects would be added.***  ***class Time:***  *"""Represents the time of a day.*  *Attributes: hour, minute, second """*  *def printTime(t): print( f’{t.hour}:{t.minute}:{t.second}’ )*  *def add\_time( t1, t2 ):*  *sum=Time()*  *sum.hour = t1.hour + t2.hour*  *sum.minute = t1.minute + t2.minute*  *sum.second = t1.second + t2.second*  *t1=Time()*  *t1.hour=10*  *t1.minute=34*  *t1.second=25*  *print("Time1 is:") printTime(t1)*  *t2=Time()*  *t2.hour=2*  *t2.minute=12*  *t2.second=41*  *print("Time2 is :")*  *printTime(t2)*  *t3=add\_time(t1, t2)*  *print("After adding two time objects:")*  *printTime(t3)* |
| ***Discuss \_\_str\_\_() and \_\_init\_\_() methods used in class definition.***  ***The \_\_str\_\_( ) Method***   * *It’s a special method that returns a string representation of an object.* * *Here we have a str method for Point object:*   *import math*  *class Point :*    *def \_\_str\_\_ (self):*  *return( f’({self.x}, {self.y})*  ***The \_\_init\_\_( ) Method***  *• A \_\_init\_\_( ) method is a special method which is similar to constructor method in other programming languages like C++/Java.*  *• The term init indicates initialization.*  *• As the name suggests, this method is invoked automatically when the object of a class is created. Consider the example given here –*    *#inside the class*  *def \_\_init\_\_(self, a, b) :*  *self.x = a*  *self.y = b*  *def dist(self, p2) :*  *d = math.sqrt((self.x - p2.x) \*\* 2 + (self.y - p2.y) \*\* 2)*  *return d*  *p1 = Point(10,20) # \_\_init\_\_ () is called automatically*  *print(p1) # \_\_str\_\_ () is called automatically*  *(10,20)*  *p2 = Point(4,5) # \_\_init\_\_ () is called automatically*  *print(p2) # \_\_str\_\_ () is called automatically*  *(4,5)*  *d=p1.dist(p2) #explicit call for dist()*  *print("Distance is:",d)*  *Distance is: 16.15549442140351* |
| *OR* |
| ***What is Encapsulation? Discuss with an example in which access specifiers are used in class definition***  *Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.*  *There are 3 types of access modifiers for a class in Python. These access modifiers define how the members of the class can be accessed.*   1. *Public* 2. *Protected* 3. *Private*   ***Public access specifier:***   * *Public members (generally methods declared in a class) are accessible from outside the class.* * *The object of the same class is required to invoke a public method.* * *This arrangement of private instance variables and public methods ensures the principle of data encapsulation.* * *By default, all the variables and member functions of a class are public in a python program.*   ***Example:***  *# defining a class Employee*  *class Employee:*  *# constructor*  *def \_\_init\_\_(self, name, sal):*  *self.name = name;*  *self.sal = sal;*  *All the member variables of the class in the above code will be by default public, hence we can access them as follows:*  *>>> emp = Employee("John", 50000);*  *>>> emp.sal;*  *50000*  ***Protected access specifier:***   * *Protected members of a class are accessible from within the class and are also available to its sub-classes.* * *Member variable can be made protected by adding \_(single underscore) to a variable name.*   *# defining a child class*  *class Research(Employee):*    *# member function task*  *def documentation(self):*  *print ("Research Documentation")*  *>>> rEmp = Research("Amanda", 100000)*  *>>> rEmp.\_sal; #Protected members*  *100000*  *>>> rEmp.documentation()*  *Research Documentation*  ***Private access specifier:***   * *While the addition of prefix \_\_(double underscore) results in a member variable or function becoming private.* * *It gives a strong suggestion not to touch it from outside the class.* * *Any attempt to do so will result in an AttributeError:*   *class Employee:*    *def \_\_init\_\_(self, name, age) :*  *self.\_\_name = name # private attribute*  *self.\_\_age = age # private attribute* |
|  |
| *Discuss polymorphism and demonstrate with and python program.*    *Polymorphism refers to having multiple forms. Polymorphism is a programming term that refers to the use of the same function name, but with different signatures, for multiple types.*  ***Example:***  *def histogram(s):*  *d = dict()*  *for c in s:*  *if c not in d:*  *d[c] = 1*  *else:*  *d[c] = d[c]+1*    *return d*  *This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, so they can be used as keys in d.* |
| Creating Classes in Python In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.  Backward Skip 10sPlay VideoForward Skip 10s  **Syntax**   1. **class** ClassName: 2. #statement\_suite   In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.\_\_doc\_\_.** A class contains a statement suite including fields, constructor, function, etc. definition.  **Example:**  **Code:**   1. **class** Person: 2. **def** \_\_init\_\_(self, name, age): 3. # This is the constructor method that is called when creating a new Person object 4. # It takes two parameters, name and age, and initializes them as attributes of the object 5. self.name = name 6. self.age = age 7. **def** greet(self): 8. # This is a method of the Person class that prints a greeting message 9. **print**("Hello, my name is " + self.name) |