MODULE-5

Classes and objects

Python Classes

A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Define Python Class

We use the `class` keyword to create a class in Python. For example,

```
class ClassName:
    # class definition
```

Here, we have created a class named `ClassName`.

Let's see an example,

```
class Bike:
    name = ""
    gear = 0
```

Here,

- `Bike` - the name of the class
- `name/gear` - variables inside the class with default values `""` and **0** respectively.

Note: The variables inside a class are called attributes.

Python Objects

An object is called an instance of a class. For example, suppose `Bike` is a class then we can create objects like `bike1`, `bike2`, etc from the class.
Here's the syntax to create an object.

```
objectName = ClassName()
```

Let's see an example,

```
# create class
class Bike:
    name = ""
    gear = 0

# create objects of class
bike1 = Bike()
```

Here, `bike1` is the object of the class. Now, we can use this object to access the class attributes

Access Class Attributes Using Objects

We use the `.` notation to access the attributes of a class. For example,

```
# modify the name attribute
bike1.name = "Mountain Bike"

# access the gear attribute
bike1.gear
```

Here, we have used `bike1.name` and `bike1.gear` to change and access the value of `name` and `gear` attribute respectively.

Example 1: Python Class and Objects

```
# define a class
class Bike:
    name = ""
    gear = 0

# create object of class
bike1 = Bike()

# access attributes and assign new values
bike1.gear = 11
bike1.name = "Mountain Bike"

print(f"Name: {bike1.name}, Gears: {bike1.gear} ")
```

Classes and functions

Time

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 16.1.

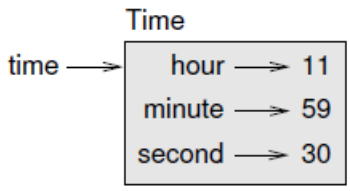


Figure 16.1: Object diagram.

Pure functions

Here is a simple prototype of add_time:

```
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
```

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.

To test this function, I'll 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
```

Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called modifiers. increment, which adds a given number of seconds to a Time object, can be written naturally

as a modifier. Here is a rough draft:

```
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
```

The first line performs the basic operation; the remainder deals with the special cases we

saw before.

In that case, it is not enough to carry once; we have to keep doing it until time.second is less than sixty. One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of increment that doesn't contain any loops.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

Debugging

A Time object is well-formed if the values of minute and second are between 0 and 60 (including 0 but not 60) and if hour is positive. hour and minute should be integral values, but we might allow second to have a fraction part. Requirements like these are called invariants because they should always be true. To put it a different way, if they are not true, something has gone wrong. Writing code to check invariants can help detect errors and find their causes. For example,

you might have a function like valid_time that takes a Time object and returns False if it

violates an invariant:

```
def valid_time(time):
if time.hour < 0 or time.minute < 0 or time.second < 0:
return False
if time.minute >= 60 or time.second >= 60:
return False
return True
At the beginning of each function you could check the arguments to make
sure they are
valid:
def add_time(t1, t2):
if not valid_time(t1) or not valid_time(t2):
raise ValueError('invalid Time object in add_time')
seconds = time_to_int(t1) + time_to_int(t2)
return int_to_time(seconds)
Or you could use an assert statement, which checks a given invariant
and raises an exception
if it fails:
def add_time(t1, t2):
assert valid_time(t1) and valid_time(t2)
seconds = time_to_int(t1) + time_to_int(t2)
return int_to_time(seconds)
```

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 the real world interact.

Printing objects

```
class Time:
"""Represents the time of day."""
def print_time(time):
print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you 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 function syntax:

```
>>> 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 the method. start is passed as a 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 method is 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 what the 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 write print_time like this:

```
class Time:
def print_time(self):
print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

Another example

Here's a version of increment (from Section 16.3) 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 a modifier.

```
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 the second parameter, seconds.

This mechanism can be confusing, especially if you make an error. For example, if you invoke increment 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 arguments in parentheses. But the subject 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 a keyword argument. In this function call:

sketch(parrot, cage, dead=True)

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

The init method

There is a method called__init___() for this task. This method is called automatically whenever a new object of a class is created. This type of function is also known as the constructor function. A constructor function is a function that is called every time a new class object is created.

Here is an example of a class Car that will clarify the functioning of__init__().

```
class Car:
    def__init__(self, name):
        self.name = name
        print("I ran first")

    def product(self):
        print("I ran second")
        return ("Name: " + self.name)


C = Car('Audi R8')
```

```
print(C.product())
```

Output:

```
I  ran  first
I ran second
Name: Audi R8
```

Explanation:

You can see in the output the order of print statements that __init___is called first.

__init___is a magic method called Dunder Methods (These are special methods whose invocation happens internally, and they start and end with double underscores). Other examples for magic methods are: __add__, __len__, __str_, etc.

Syntax of __init___function in Python

Syntax:

```
class Songs:
    def___init__(self, arguments):
        # function definition
```

__init_() function is defined just like a regular python function. Self must be the first argument. After that, you can pass arguments of your requirements. The logic of the function comes in the function definition part.

self :

self represents the instance of the class. By using the "*self*" keyword we can access the *attributes* and *methods* of the class in python.

__init___:

"__init__" is a reseved method in python classes. It is known as a constructor in object oriented concepts. This method called when an object is created from the class and it allow the class to initialize the attributes of a class.

Let's consider that you are creating a NFS game. for that we should have a car. Car can have attributes like "color", "company", "speed_limit" etc. and methods like "change_gear", "start", "accelarate", "move" etc.

```
def__init__(self, model, color, company, speed_limit):
  self.color = color
  self.company = company
  self.speed_limit = speed_limit
  self.model = model

def start(self):
  print("started")

def stop(self):
  print("stopped")

def accelarate(self):
  print("accelarating...")
  "accelarator functionality here"

def change_gear(self, gear_type):
  print("gear changed")
  " gear related functionality here"
```

Lets try to create different types of cars

```
maruthi_suzuki = Car("ertiga", "black", "suzuki", 60)
audi = Car("A6", "red", "audi", 80)
```

We have created two different types of car objects with the same class. while creating the car object we passed arguments **"ertiga", "black", "suzuki", 60 Â** these arguments will pass to *"__init__"* method Â to initialize the object.Â

Here, the magic keyword **"self" Â** represents the instance of the class. It binds the attributes with the given arguments.

Usage of "self" in class to access the methods and attributes

The__str__ method

Python str() function is used to convert an object to its string representation. It is a built-in function that can be used to convert objects of different data types, such as integers, and floats

Example:

In the given example, we assign an integer value to a variable and convert that integer variable to the string variable and print it in Python.

```
val=10
val_str= str(val)
print(val_str)
```

Python str() Function Syntax

Syntax: *str(object, encoding='utf-8?, errors='strict')*
Parameters:

- ***object:*** *The object whose string representation is to be returned.*
- ***encoding:*** *Encoding of the given object.*
- ***errors:*** *Response when decoding fails.*

Returns: *String version of the given object*

str() function in Python Example

Demonstration of str() function

In the given example, we are using str() on an empty string and string.

```
# Python program to demonstrate
# strings
```

```
# Empty string
s = str()
print(s)

# String with values
s = str("GFG")
print(s)
```

Output:

```
GFG
```

Python Operator Overloading

In Python, we can change the way operators work for user-defined types.

For example, the `+` operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called **operator overloading**.

Python Special Functions

Class functions that begin with double underscore___are called special functions in Python.

The special functions are defined by the Python interpreter and used to implement certain features or behaviors.

They are called **"double underscore"** functions because they have a double underscore prefix and suffix, such as `__init__()` or `__add__()`.

Here are some of the special functions available in Python,

Function	Description
`__init__()`	initialize the attributes of the object
`__str__()`	returns a string representation of the object
`__len__()`	returns the length of the object
`__add__()`	adds two objects
`__call__()`	call objects of the class like a normal function

Example: + Operator Overloading in Python

To overload the `+` operator, we will need to implement `__add__()` function in the class. With great power comes great responsibility. We can do whatever we like inside this function. But it is more sensible to return the `Point` object of the coordinate sum.

Let's see an example,

```
class Point:
    def___init__(self, x=0, y=0):
```

```
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)

# Output: (3,5)
```

Example: + Operator Overloading in Python

To overload the `+` operator, we will need to implement `__add__()` function in the class.

With great power comes great responsibility. We can do whatever we like inside this function. But it is more sensible to return the `Point` object of the coordinate sum.

Let's see an example,

```
class Point:
    def_init_(self, x=0, y=0):
        self.x = x
        self.y = y

    def___str__(self):
        return "({0},{1})".format(self.x, self.y)

    def___add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)

# Output: (3,5)
```

Run Code

In the above example, what actually happens is that, when we use `p1 + p2`, Python calls `p1.__add__(p2)` which in turn is `Point.__add__(p1,p2)`. After this, the addition operation is carried out the way we specified.

Type-based dispatch

The following is a version of __add__that checks the type of other and invokes either add_time or increment:

```
# inside class Time:
def__add__(self, other):
if isinstance(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 is instance takes a value and a class object, and returns True if the value is an instance of the class.

Polymorphism in Python

What is Polymorphism?

The literal meaning of polymorphism is the condition of occurrence in different forms.

Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

Let's take an example:

Example 1: Polymorphism in addition operator

We know that the `+` operator is used extensively in Python programs. But, it does not have a single usage.
For integer data types, `+` operator is used to perform arithmetic addition operation.

```
num1 = 1

num2 = 2

print(num1+num2)
```

Hence, the above program outputs `3`.

Similarly, for string data types, `+` operator is used to perform concatenation.

```
str1 =  "Python"

str2 = "Programming"

print(str1+" "+str2)
```

As a result, the above program outputs `Python Programming`.
Here, we can see that a single operator `+` has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

Function Polymorphism in Python

There are some functions in Python which are compatible to run with multiple data types.

One such function is the `len()` function. It can run with many data types in Python. Let's look at some example use cases of the function.

Example 2: Polymorphic len() function

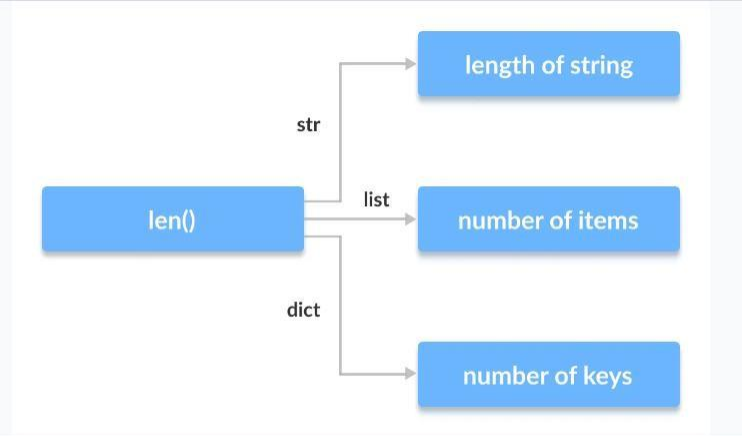
```
print(len("Programiz"))

print(len(["Python", "Java", "C"]))

print(len({"Name": "John", "Address": "Nepal"}))
```

Output

```
9
3
2
```



Polymorphism in len() function in Python

Class Polymorphism in Python

We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

We can then later generalize calling these methods by disregarding the object we are working with. Let's look at an example:

Example 3: Polymorphism in Class Methods

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")


class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Bark")


cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

Output

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method.

Another way to access attributes is the built-in function vars, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
for attr in vars(obj):
print(attr, getattr(obj, attr))
```

print_attributes traverses the dictionary and prints each attribute name and its corresponding value.

The built-in function getattr takes an object and an attribute name (as a string) and returns the attribute's value.

Interface and implementation

- One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.
- A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented.
- For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include time_to_int, is_after, and add_time.

We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a Time object are hour, minute, and second.

- As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like is_after, easier to write, but it makes other methods harder. After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface.