TONY SNAKE

PYTHON FOR ADVANCE

3 DAYS WITH PYTHON

About the Authors

Tony Snake was received Bachelor of Computer Science from the American University, and Bachelor of Business Administration from the American University, USA.

He is becoming Ph. Candidate of Department of Data Informatics, (National) Korea Maritime and Ocean University, Busan 49112, Republic of Korea (South Korea).

His research interests are social network analysis, big data, AI and robotics.

He received Best Paper Award the 15th International Conference on Multimedia Information Technology and Applications (MITA 2019)

Table of Contents

Contents

Python for Advance: 3 Days with Python

Chapter 1: Python File Handling

1. Opening and Closing Files in Python

What is File Handling in Python?

Suppose you are working on a file saved in your personal computer, if you want to perform any operating on that file like opening it, updating it or any other operation on that, that all thing come under File handling. So, File handling in computer science means working with files stored on the disk. This includes creating, deleting, opening or closing files and writing, reading or modifying data stored in those files.

Why file handling?

In computer science we have programs that do all the work. A program may require data to be read from the disk and store results on the disk for future usage. This is why we need file handling. For ex: If you need to analyse, process and store data from a website then you first need to scrap(this means getting all the data shown on a web page like text and images) the data and store it on your disk and then load this data and do the analysis and then store the processed data on the disk in a file. This is done because scraping data from a website each time you need it is not desirable as it will take a lot of time.

Opening and Closing a File in Python

In file handling we have two types of files, one is text files and other is binary files. We can open files in python using open function, let's discuss about it.

open() Function:

This function takes two arguments. First is the filename along with its complete path and the other is access mode. This function returns a file object.

Syntax:

```
open(filename, mode)
```

Important points:

- The file and python script should be in the same directory else you need to provide the full path of the file.
- By default the access mode is read mode, if you don't specify any mode. All the file opening access modes are described below.

Access mode tells the type of operations possible in the opened file. There are various modes in which we can open a

Sr. No	Modes	Description
1.	r	Opens a file in read only mode. The pointer of the file is at the beginning of the file. Thi mode.
2.	rb	Same as r mode except this opens the file in binary mode.
3.	r+	Opens the file for both reading and writing. The pointer is at the beginning of
4.	rb+	Same as r+ mode except this opens the file in binary mode.
5.	w	Opens the file for writing. Overwrites the existing file and in file is not present then c
6.	wb	Same as w mode except this opens the file in binary format.
7.	w+	Opens the file for both reading and writing, Rest is the same as w mod
8.	wb+	Same as w+ except this opens the file in binary format.
9.	a	Opens the file for appending. If the file is present then the pointer is at the end of the file file for writing.
10.	ab	Same as a mode except this opens the file in binary format.
11.	a+	Opens the file for appending and reading. The file pointer is at the end of the file if the fi a new file for reading and writing.
12.	ab+	Same as a+ mode except this opens the file in binary format.

Example of Opening and Closing Files in Python

```
# When the file is in the same folder where the python script is present. Also access mode is 'r' which is read mode.
file = open('test.txt',mode='r')

# When the file is not in the same folder where the python script is present.In this case whole path of file should be written.
file = open('D:/data/test.txt',mode='r')
```

It is general practice to close an opened file as closed file reduces the risk of being unwarrantedly updated or read. We can close files in python using close function. Let's discuss about it.

close() function:

This function don't take any argument and you can directly call close function using file object. It can be call multiple times but if any operation is performed on closed file, "ValueError" exception is raised.

Syntax:

```
file.**close()**
```

PS: You can use 'with' statement with open also as it provides better exception handling and simplifies it with by providing some cleanup tasks. Also it will automatically close the file and you don't have to do it manually.

Example with with statement:

```
with open("test.txt", mode='r') as f:
  # perform file operations
```

The method shown in the above section is not entirely safe. If some exception occurs while opening the file then the code will exit without closing the file. A more safer way is to use try-finally block while opening files.

```
try:
   file = open('test.txt',mode='r')
   # Perform file handling operations
finally:
   file.close()
```

Now this guarantees that the file will close even if an exception occurs while opening the file. So, you can also use the ‘with’ statement method instead of this. Any of the two methods is good.

Now we will see some examples on how to open and close files in python in various modes. Below is an example of a few important modes for rest you can try yourself.

The file that I will be using contains the following content.

Now we will perform some operations on the file and print the content of the file after each operation to get a better understanding how this works. The example is given below. Both the file and the python script should be in the same folder.

```
# Opening file in read mode and printing the contents of the file.
with open("test.txt", mode='r') as f:
   data = f.readlines() #This reads all the lines from the file in a list.
   print(data) #This will print content of file that is Hello World!

# Opening a file in write mode.
```

```
with open("test.txt", mode='w') as f:
    f.write("Data after write operation")
# Opening file in read mode to check the contents.
with open("test.txt", mode='r') as f:
    data = f.readlines() # this reads all the lines from the file in a list.
    print(data) #this will print the overwritten content of file that is       "Data after write operation"

# Opening a file in append mode and appending data to the file.
with open("test.txt", "a") as f:
    f.write(" Appending new data to the file")
# Opening file in read mode to check the contents.
with open("test.txt", mode='r') as f:
    data = f.readlines() #This reads all the lines from the file in a list.
    print(data) #this will print the existing content of file plus the appended content
```

Explanation

In above example firstly test.txt file was opened in read(r) mode to read its content and that file data will be printed and after that file was opened with write(w) mode, so it will overwrite all the content of that file and new data will be written to that file. After that file was opened in append(a) mode, so new data will be appended to existing data of file, will not be overwritten.

I have covered just three most important modes of file handling. You can try some other modes.on your own.

2. How to Read a File in Python

Introduction

Before we read files using python, we should first define files. To store any data, we use a contiguous set of bytes called a file. A file could be as simple as a text or an executable file. These file bytes are always converted to binary (0 and 1) format for processing by the computer.

Here are some types of files that Python can handle

1. Text files
2. Binary files

Other files that python can handle include csv files, excel sheets etc. But in this article, we will focus on the text and binary files.

What is the difference between the two files? In the text file, each line is terminated using a newline character \n. In contrast, a binary file does not

have any such terminator, and the data of the file can be stored after converting the binary file into binary language that the machine can understand.

We can now move on to reading files with the help of python.

File Access Modes

When you open a file, you want to perform an operation on it. So depending on the type of operations you'd like to perform, you select an access mode in which you can open the file.

Typically, there are six access modes:

1. **Read-only ('r')**: Denoted by r, this is the default mode for opening a file. If the file you wish to access does not exist, it raises an I/O error. With this access mode, you can only read a particular file.
2. **Write only ('w')**: This is the access mode that you use when you want to write in a file that you'd like to open. Note, that in this mode, if your file already exists, then whatever you wish to write into your file, will get written, but will delete all the initial contents of the file. When a file does not exist, a new file will be created.
3. **Append ('a')**: The append access mode allows you to write to opened files. Then what's the difference between the append and write mode? Well, in the append mode, the initial contents of the file are not deleted. Whatever you wish to write in the file will be written after the initial contents of the file.
4. **Read and Write ('r+')**: This mode allows you to read and write to a file. The handle is initially positioned at the beginning of the file. If the file does not exist, a file not found error is raised.
5. **Write and Read ('w+')**: You might be wondering the difference between read and write mode and this mode. Since this mode is an extension of the write mode, whatever you write into the file will be written after the truncation of existing contents.
6. **Append and Read ('a+')**: As the name suggests, you can read and append to a file in this access mode. Any data you wish to add will be added after the existing data.

Reading a File in Python

Now that we know the modes in which we can access a file, let's get down to reading from it. But wait, before reading it, we must first know how to open a file. And here is where we will be using our access modes.

Opening a File in Python

To open any existing or non-existing file, we use the built-in open() function. One thing to note here is that whether you're opening an existing file or you want to open a non-existing file (by creating it) depends on your access mode.

The access modes that help create a new file if it does not exist are –
w, w+, a, a+

Let’s see how we can use the open function to open an existing or non-existing file.

The syntax of the function is:

```
file_object = open(r"file_name", "access_mode")
```

The 'r' just before the parameters is to prevent the characters in the file name from being treated as special characters. For example, if we have "\tapes" in the file address, then "\t" could be treated as the tab character, and an error would be raised. The 'r' represents raw, meaning the string does not contain special characters.

The open function gives us a file object on which we can perform operations. The parameters of the open function are the file name and the access mode in which you wish to open the file. In place of the file name, if the file isn’t in the same directory as the program, we can add the complete file path.

Usage:

```
file1 = open("example.txt", "a+")
file2 = open("E:\\tests\file.txt", "w+")
```

Now, if you open a file, you must also close it.

Another important thing to note here is that when dealing with text files, it is recommended that you also specify its encoding as follows:

```
file1 = open("example.txt", "a+", encoding="utf-8")
```

Closing a File in Python

To close a file, we use the built-in function close(). This function closes the file and frees the memory space that was acquired by that file. You can use the close function if you no longer need the file or if it is required in another access mode.

Usage:

```
file1 = open("example.txt", "w+", encoding="utf-8")
# do something
file1.close()
```

That’s it!

To guarantee that the file closes, even if some exceptions are raised during file handling, a better way is to use the try … finally block. Any code in the try block is executed first. If any error is raised, it immediately stops

executing the try block code and moves to the final block.

```
try:
  file_obj = open("example.txt", "r", encoding="utf-8")
  # do something with the file
finally:
  file_obj.close()
```

Now let’s get back to business. How do we read from the file?

Naturally, if you need to read from a file in python, you would use the r mode.

To read data from a text file in python, you have 3 possible methods:

1. The read() method in python:

The read method returns a string that contains all the characters of the file mentioned. There are no mandatory parameters to this function, but if you want to read a certain number of characters from the file, you can specify them as a parameter.

Now say the file example.txt has the following contents:

Hello World
I've learnt how to use the readline() function

To read all characters:

```
file_obj = open("example.txt", "r", encoding="utf-8")
print(file_obj.read())
file_obj.close()
```

To read the first 5 characters:

```
file_obj = open("example.txt", "r", encoding="utf-8")
print(file_obj.read(5))
file_obj.close()
```

What if you wanted to read five characters but not from the current position? You want to skip the first 12 characters and print the following 5. How would you do that?

Python has a function that gives you the cursor’s current position while handling a file. And you can also ‘seek’ a certain position in the file.

```
file_obj.tell()  # will give you current cursor position
file_obj.seek(n)  # will seek the nth position
```

The read() function only allows you to either read the file's entire contents or a certain number of specified characters. What if you wanted to read the first line or the third?

2. The readline() method in python:

The readline() function helps you read a single line from the file. You can run this function as often as you require to get the information. Every time you run the function, it returns a string of the characters that it reads from a single line from the file.

If you specify an n in the parameters, it reads n number of characters from that line in the file. If n exceeds the number of characters in the file, the method returns the line without reading the following line.

Usage:

Let’s assume the contents of the file are:

Hello World
I've learnt how to use the readline() function

Code to open:

```
file_obj = open("example.txt", "r", encoding="utf-8")
print(file_obj.readline())
file_obj.close()
```

3. The readlines() method in python:

As the name suggests, the readlines() method reads all the lines in the file and returns them properly separated in a list.

Usage:

```
file_obj = open("example.txt", "r", encoding="utf-8")
print(file_obj.readlines())
file_obj.close()
```

Most of the time, you will not be using functions just like this. You might want to loop over the lines for some data you want to extract, etc. A more memory-efficient and fast way to do so is given below.

The contents of our new file are now:

This is a new file.
We have now learnt all read functions.
We are trying a new method
to loop over files.

The code to access the above contents of a file is:

```
file_obj = open("example.txt", "r")
for line in file_obj:
  print(line)
```

```
file_obj.close()
```

Reading a Binary File in Python

In the beginning of this article, we discussed two file types. In all the above examples, we've seen how to read a text file in python. All the access modes, too, were only for text files. What about binary files?

To read/write to binary files, we have special access modes: ' rbandwb. Of course, rbis for reading andwb` for writing.

It is important to note here that the return type of any operations on binary files with these access modes will be in the form of byte strings and not text strings like in text files. Similarly, when you're writing to files, the format must also be binary.

Example:

```
file_obj = open("binary.txt", "rb")
# some code
file_obj.close()
# All the functions we discussed above are applicable on the binary files as well.
```

Reading Using the with open() Method in Python

Now in all the above examples, you must have noticed that we first open the file, and then close it after we've used it. There is another way of opening files that has a much cleaner syntax and provides better exception handling. The bonus here is that you don't have to close the file after the operations you perform on it.

It is the with open() method.

Syntax:

```
with open("file_name") as f:
  data = f.read()
  # do something with this data
```

Did you notice that we did not close the file here?

Here, you open the file as f, which means that f here is the file object on which you can do operations.

You now know all you need for read files using python! So, now you can start with reading files using python.

3. How to Write a File in Python

Writing to a text file in Python

We have various access modes to open an existing text file in Pyhton and to write to it, depending on what we need. Even if the text file does not already exist, we can use the w or the a access mode to create a text file and then write to it. There are two functions in Python that help us write into text files:

1. The write() function in Python:

Whatever strings you give as a parameter to this function, it writes it as a single line in the text file. Now again, whether or not the existing contents will be truncated depends on the access mode. If you use the w mode, the contents will be truncated, and your string will be written. However, in the a mode, your existing contents will not be deleted, and your string will be written after the contents.

Now let's take a look at an example. We initially do not have any existing file with the name of writing.txt. We're creating it using the w mode. Once we open the new file, it's obviously empty. We're then going to write content into it.

```
file_obj = open("writing.txt", "w")
```

Here, file_obj is the file object. We're opening a file named writing.txt in the w access mode. If this file does not exist, since we have used the w access mode, it will be created.

We can now write into it like this:

```
file_obj.write("This is how you write to a file")
file_obj.write("Now this will be written to another line")
```

Since we used the open() method to open the file, instead of with open(), we need to close the file as well.

Let’s see what our file looks like now with the below program.

```
{"output":"This is how you write to a file /n
Now this will be written to another line "}
print(file_obj.read())
file_obj.close()
```

The non-existent file now has above two lines of content in it. Let's try writing to the file again, just another line. Since we closed the file earlier, we need to open it again.

```
file = open("writing.txt", "w")
```

Note how we used the w access mode. The file writing.txt currently exists. The previous time we used the open function to open the file, it did not exist. We created it and then opened it. This time, it just opens in writing mode since it already exists.

```
file.write("Where do you think this will be written in the file?")
file.write("Obviously, the initial contents will be overwritten with these two lines")
print(file.read())
file.close()
```

Once you run this code, you see the below file contents:

```
Where do you think this will be written in the file?
Obviously, the initial contents will be overwritten with these two lines.
```

To keep the initial contents of the file, you will have to use the append (a) access mode. Since you will change the access mode, you need to re-open the file in that mode after closing it.

```
file = open("writing.txt", "a")
file.write("This way, I will preserve the existing contents in the file")
print(file.read())
file.close()
```

Output

```
Where do you think this will be written in the file?
Obviously, the initial contents will be overwritten with these two lines
This way, it will preserve the existing contents in the file
```

2. The writelines() function in Python:

Constantly using file.write() for every line that we want to write to your file can get difficult. Hence, we can use the writelines() function.

A simple way to use it, is to provide a list of strings as a parameter to writelines().

```
lines = ["line1", "line2", "line3"]
file_obj.writelines(lines)
```

Note that if you open a file with w or a access mode, you can only write to the file and not read from it. In the same way, if you open a file in the r mode, you can only read from it and not write. If you wish to perform both operations simultaneously, you should use the a+ mode.

Writing to a binary file in Python

After learning how to read data from a binary file, I'm sure you know how to write to it too. We use the wb mode to write to a binary file.

Example

```
f = open("binfile.bin", "wb")
nums = [1, 2, 3, 4, 5]
arr = bytearray(nums)
f.write(arr)
f.close()
```

Obviously, binary data is not human recognizable. So when we have to write an array of numbers like 1, 2, 3, 4, and 5, we need to first convert them into a byte representation to store it in the binary file. For that purpose, we use the built-in bytearray() function.

4. How to Delete File in Python?

Introduction

Sometimes after you've worked with your file and do not require it anymore, you can delete it using python. Say, in an extensive program, you needed to create files to store data but did not require them after the program's execution. It's a good idea to delete those files, and we'll see how to do that using python in this article.

How to Delete Files Using Python?

Removing the files or a single file from a particular directory when it is no longer required is the basic concept of deleting a file. Now to remove a file, you have three methods. Using one of the modules:

1. os
2. shutil
3. pathlib

1. Using the os module in python

The os module allows you to use the operating system-dependent functionalities.

To use the os module to delete a file, we first need to import it, then use the remove() function provided by the module to delete the file. It takes the file path as a parameter.

You can not just delete a file but also a directory using the os module.

To delete a file:

```
import os
file_path = <file_path>
if os.path.isfile(file_path):
  os.remove(file_path)
  print("File has been deleted")
else:
  print("File does not exist")
```

In this piece of code, we're first importing the os module and then storing the complete file path of the file we wish to delete in a variable called file_path. We then check if the file exists at that path, then we remove the file, and if it doesn't exist, then we do nothing.

On the other hand, if you wish to delete or clear a directory, you can do it with the help of the rmdir() function of the os module. Note that the directory must be empty for this to work.

```
import os
os.rmdir('directory')
```

If the directory to delete is in the same directory as the python program, then you need not provide a full path. A relative path will work. Else, a path can be written as the parameter of the rmdir function.

2. Using the shutil module in python:

The shutil module is a high-level file operation module. You can perform functions like copying and removal on files and collections of files.

This module can also be used as the os module to delete files or a directory. But here, the directory is not necessary to be empty. If you delete a directory using shutil, you can also delete all the contents inside of it (files and subdirectories).

The function of the shutil module is rmtree().

```
import shutil
shutil.rmtree('path')
```

Now here is the place of path you can provide the path to your directory that you wish to remove.

You cannot delete a single file with the shutil.rmtree() function.

3. Using the Pathlib Module in Python

If you're working with a python version 3.4+, then the pathlib module is beneficial for deleting/removing files.

The pathlib module has many similarities with the os module, two of them being the remove and rmdir methods.

Now, when working with this module, you must first create a Path object. When an instance of the Path class is created,
a WindowsPath or PosixPath will be returned according to the machine you're working on. For Windows OS, a WindowsPath object will be returned, and for non-windows OS like linux, PosixPath will be returned.

```
import pathlib
p_object = Path(".")
type(p_object)
```

Then, the next step is to use the unlink() function. The unlink() function is used to remove the file or the symbolic link. If you wish to delete a directory, you must use the rmdir() function instead.

```
import pathlib
file = pathlib.Path("test/file.txt")
file.unlink()
```

The same way, if you want to delete a directory:

```
import pathlib
directory = pathlib.Path("files/")
directory.rmdir()
```

5. With Statement in Python

What is "with" Statement in Python

Anything that happens in programming consumes memory and disk space. When we create software we store it into the hard drive and run it in the memory. One problem that occurs is, if the programmer does not manage these resources properly then it causes issues like, the software will be slow, consume too many resources, etc.

One common scenario where this happens is when we try to work with files. Whenever a file is opened and an operation like writing text is performed, the data is stored in the temporary buffer.

Wait… What is a buffer?

Buffer is a place to store something temporarily. This is done when the input

and output speed are different. A simple example is, when you watch any video online, the video service downloads the next few minutes of content to avoid any interruptions.

Buffer is a generic term and it is not specific to tech.

Coming back to the topic, when we open any file, our computer stores the file into a buffer so that we are able to perform operations on it without any issues.

Let's say we open a file with python code, we perform operations on it, but we forget to close the file, then all the operations that were performed will be lost. Why? The reason is until the file is saved all the changes are made to the data stored in the buffer, and if we do not save the file then our computer will think that is unuseful data and will remove it. In simple words, our changes will be lost.

A similar thing can happen when there are exceptions and errors in the program.

Try…Finally in Python

This is a simple approach, we "**try**" to do something (performing operations), and then we do some other thing "**finally**".

It doesn't matter if the code inside **"try"** is successful or not, code inside **"finally"** will run after the try block.

This is a very common practice in python when we work with files. We open a file, do some operations like write to the file inside a **"try"** block and then close the file in the **"finally"** block.

With in Python

with statement in python aims to simplify the process of try…finally. It enables us to write cleaner and safer code. with can be considered as a simplified version for try…finally code. It handles the memory allocation and deallocation automatically.

Most commonly with is used with files

```
with open('hello.txt', mode='w') as myFile:
    myFile.write('I am inside with')
```

In the above code, we use with to open a file and perform operations on it. When the **with** block is executed it automatically closes the file even when an exception is raised.

If **with** is not used then it is the responsibility of the programmer to close the file. If the file is not closed then the performed operations will not be saved on the file.

But how does “with” know to close a file?

with statement can be used with all expressions that return an object which implements the context management protocol. Context managers allows us to allocate and release resources when we want to. This protocol contains two special functions:

1. __enter__()
2. __exit__()

As the name suggests, **__enter()__** is called when the control enters **with** block, and **__exit()__** is called when the control leaves **with** block.

When **with** statement executes, __enter__() is called on the context manager object to indicate the starting and when the flow leaves the context, __exit__() is called to indicate the end.

NOTE: with is shorter than try…finally but it is also less general which means **“with”** will work only with objects that support **context management protocol.**

Using “with” in user-defined objects

“with” has support with a lot of built-in methods like **open()**(used for opening files)but we can also use it in our custom methods. To use it in our custom methods we have to follow some conditions of with, let’s see how it can be done.

Wait… Why would I use “with” in my custom function?

“With” statement can help in many tasks that need to be performed at the start or end of a procedure.

E.g. In the case of file handling, you can define methods that can open and close the file for you. Or you are creating an app where you need to contact with a database, there you need to start a connection when the data transfer takes place and end the connection when the transfer is done.

“with” has a condition that it works with objects that support **context**

management protocol, or it works with objects that have an **__enter()__** and **__exit()__** method.

We can make our own class with these methods and that will enable us to use **“with”.**

Here is an example of a simple class.

```
class MyContextManager:
    def __enter__(self):
        print("Entered the context")
        return "Entered!"

    def __exit__(self, *args, **kwargs):
        print("Leaving the context")


with MyContextManager() as hello:
    print(hello)
```

Output:

```
Entered the context
Entred!
Leaving the context
```

In the above code, we created a class **MyContextManager** and created two methods inside it named __enter__() and __exit__(). When we call this class using **“with”** we can see that the __enter__() runs first and then __exit__() runs.

In the code *args and **kwargs are used because __exit__() takes three arguments

- execution_type
- execution_value
- traceback

These arguments should occur in this order.

- **execution_type** is exception’s class.
- **execution_value** is an exception instance which indicates the type of exception like, Zero Division Error, Floating Point Error
- **traceback** is traceback object

When there are no exceptions raised, then all these args will return **None, None, None.**

Contextlib Library

contextlib library has a **“contextmanager”** decorator which provides an alternative and simple way to implement context management protocol. This reduces a lot of boilerplate code as it automatically provides __enter()__ and __exit()__ methods.

Let’s look at an example.

```
from contextlib import contextmanager


@contextmanager
def myContextManager():
    print("Entering...")
    yield "Hello There"
    print("Exiting...")


with myContextManager() as cm:
    print(cm)
```

Output:

```
Entering...
Hello There
Exiting...
```

Chapter 2: Python Exception Handling

1. Exception Handling in Python

Introduction

Writing code in order to solve or automate some huge problems is a pretty powerful thing to do, isn't it? But as Peter Parker says, 'with great power comes great responsibility'. This in fact holds true when writing your code as well! While our code is being executed, there might be some events that disrupt the normal flow of your code. These events are errors, and once these occur python interpreter is in a situation that it cannot deal with and hence raises an **exception**.

In python, an exception is a class that represents error. If these exceptions are not handled, our application or programs go into a crash state. As a developer, we definitely have the power to write code and solve problems but it becomes our responsibility to handle these exceptions that might occur and disrupt the code flow.

Let's see how these exceptions can be handled in this article but first of all, let's take a look at an example that will introduce you to an exception:

```
a = 10
b = 0
c = b/a
print(c)
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    c = a/b
ZeroDivisionError: division by zero
```

Above is an example of what we call an unhandled exception. On line 3, the code went into a halt state as it's an unhandled exception. To avoid such halt

states, let’s take a look at how to handle exceptions/errors.

What is Exception Handling in Python?

Exceptions can be unexpected or in some cases, a developer might expect some disruption in the code flow due to an exception that might come up in a specific scenario. Either way, it needs to be handled.

Python just like any other programming language provides us with a provision to handle exceptions. And that is by try & except block. Try block allows us to write the code that is prone to exceptions or errors. If any exception occurs inside the try block, the except block is triggered, in which you can handle the exception as now the code will not go into a halt state but the control will flow into the except block where it can be manually handled.

Any critical code block can be written inside a try clause. Once the expected or unexpected exception is raised inside this try, it can be handled inside the except clause (This ‘except’ clause is triggered when an exception is raised inside ‘try’), as this except block detects any exception raised in the try block. By default, except detects all types of exceptions, as all the built-in exceptions in python are inherited from common class Exception.

The basic syntax is as follows:

```
try:
   # Some Code
except:
   # Executed if we get an error in the try block
   # Handle exception here
```

Try and except go hand in hand i.e. the syntax is to write and use both of them. Writing just try or except will give an error.

Let’s consider an example where we’re dividing two numbers. Python interpreter will raise an exception when we try to divide a number with 0. When it does, we can take custom action on it in the except clause.

```
def divideNos(a, b):
  return a/b

try:
  divideNos(10, 0)
except:
  print('some exception occured')
```

Output:

```
some exception occured
```

But how do we know what exactly error was read by the python interpreter?

Well, when a python interpreter raises an exception, it is in the form of an object that holds information about the exception type and message. Also, every exception type in python inherits from the base class Exception.

```
def divideNos(a, b):
  return a/b
try:
  divideNos(10, 0)
# Any exception raised by the python interpreter, is inherited by the base class 'Exception', hence any exception raised in the try block will be detected and collected further in the except block for handling.
except Exception as e:
 print(e) # as e is an object of type Exception, Printing here to see what message it holds.
 print(e.__class__)
division by zero
<class 'ZeroDivisionError'>
```

In the above code, we used the Exception class with the except statement. It is used by using the as keyword. This object will contain the cause of the exception and we're printing it in order to look what the reason is inside this Exception object.

Common Exceptions in Python

Before proceeding further, we need to understand what some of the common exceptions that python throws are. All the inbuilt exceptions in python are inherited from the common 'Exception' class. Some of the common inbuilt exceptions are:

Exception Name	Description
Exception	All exceptions inherit this class as the base class for all exceptior
StopIteration	Raised when the next() method of an iterator while iteration does not point
StandardError	All exceptions inherit this class except stop StopIteration and Syster
ArithmeticError	Errors that occur during numeric calculation are inherited by it
OverflowError	When a calculation exceeds the max limit for a specific numeric data
ZeroDivisionError	Raised when division or modulo by zero takes place.
AssertionError	Raised when an assert statement fails
AttributeError	Raised in case of failure of attribute assignment
EOFError	Raised when the end of file is reached, usually occurs when there is no input fror
ImportError	Raised when an import statement fails
NameError	Raised when an identifier is not found in the local or non-local or globa
SyntaxError	Raised when there is an error in python syntax.
IndentationError	Raised when indentation is not proper
TypeError	Raised when a specific operation of a function is triggered for an invalid

ValueError	Raised when invalid values are provided to the arguments of some builtI for a data type that has a valid type of arguments.
RuntimeError	Raised when an error does not fall into any other category
NotImplementedError	Raised when an abstract method that needs to be implemented in an ir class is not actually implemented.

Catching Specific Exceptions in Python

In the above example, we caught the exception that was being raised in the try block, but the except blocks are going to catch all the exceptions that try might raise.

Well, it's considered a good practice to catch specific types of exceptions and handle them accordingly. And yes, try can have multiple except blocks. We can also use a tuple of values in an except to receive multiple specific exception types.

Let's take an example to understand this more deeply:

```
def divideNos(a, b):
  return a/b # An exception might raise here if b is 0 (ZeroDivisionError)
try:
   a = input('enter a:')
   b = input('enter b:')
   print('after division', divideNos(a, b))
   a = [1, 2, 3]
   print(a[3]) # An exception will raise here as size of array 'a' is 3 hence is accessible only up until
2nd index

# if IndexError exception is raised
except IndexError:
   print('index error')
# if ZeroDivisionError exception is raised
except ZeroDivisionError:
   print('zero division error')
```

Output:

```
enter a:4
enter b:2
after division 2.0
index error
```

- In the above code the exceptions raised totally depend upon the input that the user might enter. Hence if a user enters a value as 0 for 'b', the python interpreter will raise a ZeroDivisionError.
- And as the array 'a' has a length of 3, and the user is trying to access an element at index 3, an IndexError will be raised by the python

interpreter.

- Each except block has been defined for both the exceptions as one of them receives exceptions of type IndexError and the other receives of type ZeroDivisionError.

If we want the above snippet to catch exception for both IndexError OR ZeroDivisionError, it can be re-written as:

```
def divideNos(a, b):
 return a/b
try:
a = int(input('enter a:'))
b = int(input('enter b:'))
 print('after division', divideNos(a,b))
a = [1, 2, 3]
 print(a[3])
except (IndexError, ZeroDivisionError):
  print('index error OR zero division error')
```

Output:

```
enter a:10
enter b:2
after division 5.0
index error OR zero division error
```

Note: Only one of the except blocks is triggered when an exception is raised. Consider an example where we have one except as except IndexError and another as except(IndexError, ZeroDivisionError) then the one written first will trigger.

Raising Custom Exceptions

Even though exceptions in python are automatically raised in runtime when some error occurs. Custom and predefined exceptions can also be thrown manually by raising it for specific conditions or a scenario using the raise keyword. (A custom exception is an exception that a programmer might raise if they have a requirement to break the flow of code in some specific scenario or condition) String messages supporting the reasons for raising an exception can also be provided to these custom exceptions.

Syntax to Raise an Exception

```
try:
   # on some specific condition or otherwise
   raise SomeError(OptionalMsg)
Except SomeError as e:
```

```
  # Executed if we get an error in the try block
  # Handle exception ‘e’ accordingly
```

For example:

```
def isStringEmpty(a):
  if(type(a)!=str):
     raise TypeError('a has to be string')
  if(not a):
     raise ValueError('a cannot be null')
  a.strip()
  if(a == ''):
     return False
  return True

try:
a = 123
 print('isStringEmpty:', isStringEmpty(a))
except ValueError as e:
  print('ValueError raised:', e)

except TypeError as e:
  print('TypeError raised:', e)
```

Output:

```
TypeError raised: a has to be string
```

- In the above code, the variable ‘a’ can hold whatever value that is assigned to it. Here we assign it a number and we’re passing to a custom method isStringEmpty that checks if a string is an empty string.
- But we orchestrated it to throw a TypeError, by assigning ‘a’ variable a number.
- In the method, we’re checking if the variable is a string or not and if it holds a value or not. In this case, it is supposed to be a string, but assigned it as a number as we’re raising an exception by using

try except and ELSE!

Sometimes you might have a use case where you want to run some specific code only when there are no exceptions. For such scenarios, the else keyword can be used with the try block. Note that this else keyword and its block are optional.

Syntax With Else Clause

```
try:
    # on some specific condition or otherwise
    raise SomeError(OptionalMsg)
except SomeError as e:
    # Executed if we get an error in the try block
    # Handle exception ‘e’ accordingly
else
    # Executed if no exceptions are raised
```

Example:

When an exception is not raised, it flows into the optional else block.

```
try:
b  = 10
c = 2
a = b/c
 print(a)
except:
 print('Exception raised')
else:
  print('no exceptions raised')
```

Output:

```
5.0
no exceptions raised
```

In the above code, As both the inputs are greater than 0 which is not a risk to DivideByZeroException, hence try block won’t raise any exception and hence ‘except’ block won’t be triggered. And only when the control doesn’t flow to the except block, it flows to the else block. Further handling can be done inside this else block if there is something you want to do.

Example:

When an exception is raised, control flows into the except block and not the else block.

```
try:
   b  = 10
   c = 0
   a = b/c
   print(a)
except Exception as e:
   print('Exception raised:', e)
else:
   print('no exceptions raised')
```

Output:

```
Exception raised: division by zero
```

In the above code, As both the ‘b’ input is 0 which is a risk to DivideByZeroException, hence the ‘try’ block will raise an exception, and hence the ‘except’ block will be triggered. And now as there is an exception raised, the control flows to the except block and not to the else block.

Try Clause with Finally

Finally is a keyword that is used along with try and except, when we have a piece of code that is to be run irrespective of if try raises an exception or not. Code inside finally will always run after try and catch.

Example:

Where an exception is raised in the try block and except is triggered.

```
try:
temp = [1, 2, 3]
temp[4]
except Exception as e:
 print('in exception block: ', e)
else:
 print('in else block')
finally:
 print('in finally block')
```

Output:

```
in exception block: list index out of range
in finally block
```

In the above code, we’re creating an array with 3 elements, i.e. max index up till 2. But when we try to access the 4th index, it will raise an exception of index out of range and will be caught in the except block. But here we need to observe that the finally block has also been triggered.

Example:

Rewriting the above code such that exception is not raised. Where an exception is not raised and else and finally both are triggered.

Note: else block will always be triggered before finally and finally will always trigger irrespective of any exceptions raised or not.

```
try:
temp = [1, 2, 3]
temp[1]
except Exception as e:
 print('in exception block: ', e)
else:
 print('in else block')
```

```
finally:
 print('in finally block')
```

Output:

```
in else block
in finally block
```

In the above code, we're creating an array with 3 elements, i.e. max index up till 2. But when we try to access the 2nd index, now it will not raise an exception and control will now flow to another block and then to finally.

But here we need to observe that the finally block has also been triggered even though the exception was not raised.

Why Use Finally or Else in try..except?

```
def Checkout(itemsInCart):
  try:
    id = GetOrderIdForBooking() # Always creates an order before booking items
    BookItems(itemsInCart) # assuming this method raises an exception while booking one of the items
  except Exception as e:
    LogErrorOccuredWhileBooking(e) # Log failed booking
  else:
    LogSuccessfulBookingStatus(id) # Log successful booking
  finally:
    EmailOrderStatusToUser(id) # Either ways send order email to user

itemsInCart = ['mobile', 'earphones', 'charger'] # assume user has these items in their cart
Checkout(itemsInCart) # checking out items the user has in their cart.
```

- Consider above as a pseudo-code to a program where it checks out items added by the user in their cart.
- The Checkout method always creates an order id before booking all the items in the cart.
- Assuming itemsInCart contains all the items the user has added to a cart.
- In the checkout experience of an e-commerce website, an orderId is created irrespective of whether a booking of all items has been a success, failed, or partially failed.
- Hence in the above pseudo-code, we're creating an orderId using GetOrderIdForBooking() and assume it returns a random string for each order. BookItems() books all the items for us in the itemsInCart array.
- If some error occurs while booking items, we have to log the reason for

the exception LogErrorOccuredWhileBooking() does that job for us and if it's successful we have to log that the booking was successful.

- Now whatever the status of the order is, either way we have to email the status of the order to the user and that's what the EmailOrderStatus() does for us.

The above example perfectly fits for an application of try, except or else and finally block as each block is playing a role that might've been difficult to achieve if we didn't use these blocks.

What Happens if Errors are Raised in Except or Finally Block?

```
try:
  SomeFunction() # assuming it throws some error
except exception as e:
  Log(e) # assuming some exception occurred again while logging this exception
finally:
  # This will still execute even after an error was raised inside catch block & gives you a final chance to handle the situation your way
```

- If an error occurs while handling something in an except block, the finally block still gives us a chance to handle the situation our way. **Note:** Once an exception/error is raised in the except block, finally block is triggered but the program still goes into a halt state post to that and the flow is broken.
- But what if an error occurs in the finally block? Let's orchestrate these series of events in the following code:

```
try:
  raise Exception("message from try block")
except Exception as e:
  print('in except block:', e)
  raise Exception("exception raised in except block")
finally:
  print("in finally")
  raise Exception("exception raised in finally block")
```

Output:

```
in except block: message from try block
in finally
Traceback (most recent call last):
  File "<string>", line 2, in <module>
Exception: message from try block
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<string>", line 5, in <module>
Exception: exception raised in except block
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<string>", line 8, in <module>
Exception: exception raised in finally block
```

2. Assert Keyword in Python

Introduction to Assert in Python

Multiple times during programming, the programmer knows that there are certain conditions or assumptions that are always going to be True. If at any point of time during the execution of the program, any statement with the condition that is meant to be True evaluates to False, then it shouldn't allow the code to execute further. Essentially, we immediately terminate the program as soon as we find something fishy, that shouldn't exist. This task is achieved with the help of assert.

What is Assert?

An assert statement is something that can be used to verify any logical assumptions you make in your code. It is mainly used as a debugging tool. For example, if we write a function that performs division, we know that the divisor must not be zero, so we "assert" that the divisor shouldn't be equal to zero so as to prevent any kinds of errors/bugs that could occur due to this problem in the code later on.

As input, the assert in python takes an expression that is to be evaluated, and an error message, which is optional. Basically, when the code encounters any assert statements, one of two things can happen.

One, the code runs without any issues, and two, it throws an AssertionError, terminating the program. We are going to delve deep into these situations, but we must first understand why exactly the assert statement should be used.

Why Use Assert in Python?

An assert statement recognizes issues right off the bat in your program, where the reason is clear, instead of later when some other operation fails.

They can be termed as internal self-checks for your code. The aim of using the assert statements is so that developers can track the root cause of the bug in their code quickly.

Assert statements particularly come in handy when testing or assuring the quality of any application or software.

Python Assert Statement

Syntax:

```
assert <condition> , <error message>
```

As stated above, we have a condition and an error message as part of the assert statement. If the condition used in the assert statement evaluates to True, the code runs normally i.e. without throwing errors or terminating the program. On the other hand, if it evaluates to False (unwanted condition) it throws an AssertionError along with the error message, if provided and terminates the program.

Assert in python,

```
assert <condition>
```

is almost similar to this:

```
if __debug__:
   if not <condition>:
      raise AssertionError()
```

What is __debug__ here? It is a constant that is used by Python to determine if statements like assert must be executed.

In the Python programming language, whether we use the assert statement, or raise the error using if, it does not have any major differences, they’re almost similar. The code with if raises an AssertionError when the debug mode is on (__debug__ == True), and the condition provided evaluates to False.

When we can use an if statement, which is definitely easier to understand, **why** boggle minds with the assert statement? There are two main reasons – readable code, and the option of disabling the assert statements if required.

Using the assert statement in place of the if condition makes the code more

readable and shorter. An advantage of using the assert statement over the if condition is that an assert statement can be disabled, unlike an IF condition.

To disable any assert statements in the program, running python in optimized mode (__debug__ == False), ignores all assert statements. The -O flag can be passed for the same.

python -O program.py

Now that we have seen the syntax of the assert statement, as well as its advantage over if, let us move on to see what it looks like in the Python shell, and how we run it:

```
>>> assert True   # nothing, because <condition> is True
>>> assert False  # will raise an error but no message, since not specified
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert False, "This statement is False"  # condition is False, hence error + message, since it is specified

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: This statement is False
```

If the assertion fails, a message can be printed like this as done in the shell example above :

```
assert False "This assertion has failed"
```

Not adding the error message does not make any difference in the way the program runs. The error displayed will be the Traceback error as shown in the example above and terminates the program. It just helps understand the assertion error more clearly.

Assertion errors terminate the program you have written, but, what if we did not want that to happen? What if we want the code to run as-is, and along with it, we would like the error to be detected?

The solution for this problem is the try-except block. The try-except block in Python is another means of handling and checking errors in Python. First, only the code that is written inside the try block will get executed when there isn't any error in the program and the except clause is skipped. While in the presence of an error in the try block, the code in the except block will be executed and code in the try block will be skipped.

Here's how we handle a basic division by 0 assertion error using the try-

except block:

```
try:
  x = 1
  y = 0
  assert y != 0 , "Division by zero not possible!"
  print(x / y)
except for AssertionError as msg:
  print(msg)  # error message given by user gets printed

print("The rest of the program still runs!")
```

Let’s try and understand what happened in the above code. We used an assert statement, that, as we read above prints an error message upon evaluation of condition to False, and we also print a msg in the except block. What are we doing here, and why?

When we’re in the try block, we have values of x as 1 and y as 0, and an assert condition that prevents the division of any number by 0. Now since the value of y is 0, our assert statement evaluates to False and hence throws an AssertionError. We know that these errors terminate the program, but since we do not want it to terminate, we use the try-except. As soon as there is an error in the try block, we immediately move to the except block to handle the errors.

When we write a simple assert statement with a condition and an error message, what gets printed? The error message. So, the except block

```
>>> assert(2 + 3 == 6, "Incorrect addition")  # wrong
>>> assert 2 + 3 == 6, "Incorrect addition"   # correct
```

The first statement with the parentheses will not work because you’re essentially providing the assert statement with a tuple. The format of a tuple in python is (Val1, Val2, …) and any tuple written in the context of bool is always true unless it does not contain any values. This means that if we do not provide any condition to the assert statement, like :

assert()

it will raise an AssertionError because empty tuples are False.

What Does the Assertion Error do to our Code?

Upon reading and understanding multiple times that unhandled assertions terminate programs, a question comes to mind – How will it terminate the program?

There are essentially two ways of stopping the execution of a program –

by exit() and by abort().

When a program is terminated by using exit(), the function performs – flushing of unwritten buffered data, closing of open files, removal of temporary files and it returns an integer exit status to the operating system.

The abort() function may not close any files that are open, may not flush stream buffers, and may also not delete any temporary files. One major difference between both kinds of functions that terminate a program is that abort() results in abnormal termination while exit() causes normal termination.

Termination of the program due to an assert statement occurs with the help of the abort() function when the condition mentioned in the assert statement returns False.

Finally, now that we know what assert is, why it is useful, and how we can tackle it, we must address the elephant in the room. Where exactly can you use it?

Where is Assertion used in Python?

Assertions are not used to flag expected error conditions, similar to “file not found” where a user can make a restorative move (or simply attempt once more). Instead, the assert statement is used mainly for debugging purposes,

1. Checking parameter types / classes / values

The assert statement can be used if your program has the tendency of controlling every parameter entered by the user or whatever else

Example:

```
def kelvin_to_fahrenheit(Temp):
  assert (Temp >= 0),"Colder than absolute zero!"  # User input cannot violate a fact, hence the use of assert
  return ((Temp - 273) * 1.8) + 32
```

2. Checking “can’t happen” situations

While the assert statement is useful in debugging a program, it is discouraged when used for checking user input unless it corrupts a universal truth. The statement must only be used to identify any “this should not happen” situation i.e. a situation where any fact, or a statement that is definitely True, becomes False due to a bug in the code.

For example, if the user enters fewer characters than required in the input, then it cannot be termed as a “this should not happen” situation, where the

use of the assert statement is a must because the user isn't violating any fact that is meant to be True always. It is not a blunder to use an assert statement for trivial issues such as this, but it is considered a bad practice.

```
def something(s: str):  # a function that takes a long string as the input to do something
  assert len(s) >= 8  # bad practice, the length of the input string doesn't violate a universal truth

  # do something
```

However, if say, a program involves calculations with radius and diameter of a circle and the radius of the circle isn't half of the diameter at any point in the program, then this comes under the "this should not happen" case, as it is a fact that cannot be changed or violated.

For example,

```
# some complex operations involving the radius of a circle

assert radius == diameter / 2, "Calculations have corrupted the value of radius!!!"

# return statements with radius or continuation in the program
```

If during the complex operations, the value of the radius changes and is unequal to half of the diameter, our assert statement condition immediately detects it and throws an error so as to avoid any further complications/errors in the code or provide unwanted output.

3. Documentation of Understanding

Using the assert statement in your code is useful for documenting the programmer's understanding of the code. It makes the programmer's assumptions clear for themselves, as well as anyone else who works on the same code in the future. If for any reason, in the future the code gets broken, the reason for failure will be obvious, and the next time the assert condition evaluates to False, the caller will get the AssertionError exception.

Another Example of Assertion in Python

Say you have a function that calculates the cost of an item after a discount. In this situation, we can positively say that the discounted cost will definitely be greater than 0 and lesser than the original cost.

If the discounted cost doesn't lie in this range, then we are in a "this should not happen" situation. Hence, we use the assert statement.

```
def calculate_discount(cost, discount):
    discounted_cost = cost - [discount * cost]
    assert 0 <= discounted_cost <= cost  # used an assert because a fact could have been violated, and also serves as a depiction of the understanding of the programmer
```

```
return discounted_cost
```

Chapter 3: Python Object & Class

1. Python Object Oriented Programming

Object Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

A parrot is an object, as it has the following properties:

- name, age, color as attributes
- singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can

study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:

    pass
```

Here, we use the `class` keyword to define an empty class `Parrot`. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, `obj` is an object of class `Parrot`.
Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example 1: Creating Class and Object in Python

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
```

```
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name `Parrot`. Then, we define attributes. The attributes are a characteristic of an object.
These attributes are defined inside the `__init__` method of the class. It is the initializer method that is first run as soon as the object is created.
Then, we create instances of the `Parrot` class. Here, `blu` and `woo` are references (value) to our new objects.
We can access the class attribute using `__class__.species`. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using `blu.name` and `blu.age`. However, instance attributes are different for every instance of a class.

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
```

```
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Output

```
Blu sings 'Happy'
Blu is now dancing
```

In the above program, we define two methods i.e `sing()` and `dance()`. These are called instance methods because they are called on an instance object i.e `blu`.

Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
```

```
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

In the above program, we created two classes i.e. `Bird` (parent class) and `Penguin` (child class). The child class inherits the functions of parent class. We can see this from the `swim()` method.

Again, the child class modified the behavior of the parent class. We can see this from the `whoisThis()` method. Furthermore, we extend the functions of the parent class, by creating a new `run()` method.

Additionally, we use the `super()` function inside the `__init__()` method. This allows us to run the `__init__()` method of the parent class inside the child class.

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single `_` or double `__`.

Example 4: Data Encapsulation in Python

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

Output

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a `Computer` class.
We used `__init__()` method to store the maximum selling price of `Computer`.
Here, notice the code

```
c.__maxprice = 1000
```

Here, we have tried to modify the value of `__maxprice` outside of the class. However, since `__maxprice` is a private variable, this modification is not seen on the output.
As shown, to change the value, we have to use a setter function i.e `setMaxPrice()` which takes price as a parameter.

Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

Example 5: Using Polymorphism in Python

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

Output

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes `Parrot` and `Penguin`. Each of

them have a common `fly()` method. However, their functions are different. To use polymorphism, we created a common interface i.e `flying_test()` function that takes any object and calls the object's `fly()` method. Thus, when we passed the `blu` and `peggy` objects in the `flying_test()` function, it ran effectively.

Python Objects and Classes

Python Objects and Classes

Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.

An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

We can think of a 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.

As many houses can be made from a house's blueprint, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

Defining a Class in Python

Like function definitions begin with the def keyword in Python, class definitions begin with a class keyword.
The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
```

```
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
There are also special attributes in it that begins with double underscores __. For example, __doc__ gives us the docstring of that class.
As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

Output

```
10
<function Person.greet at 0x7fc78c6e8160>
This is a person class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>> harry = Person()
```

This will create a new object instance named harry. We can access the attributes of objects using the object name prefix.
Attributes may be data or method. Methods of an object are corresponding functions of that class.

This means to say, since Person.greet is a function object (attribute of class), Person.greet will be a method object.

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# create a new object of Person class
harry = Person()

# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(harry.greet)

# Calling object's greet() method
# Output: Hello
harry.greet()
```

Output

```
<function Person.greet at 0x7fd288e4e160>
<bound method Person.greet of <__main__.Person object at 0x7fd288e9fa30>>
Hello
```

You may have noticed the self parameter in function definition inside the

class but we called the method simply as `harry.greet()` without any arguments. It still worked.
This is because, whenever an object calls its method, the object itself is passed as the first argument. So, `harry.greet()` translates into `Person.greet(harry)`.
In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called `self`. It can be named otherwise but we highly recommend to follow the convention.
Now you must be familiar with class object, instance object, function object, method object and their differences.

Constructors in Python

Class functions that begin with double underscore `__` are called special functions as they have special meaning.
Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.
This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')


# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)

# Call get_data() method
# Output: 2+3j
```

```
num1.get_data()

# Create another ComplexNumber object
# and create a new attribute 'attr'
num2 = ComplexNumber(5)
num2.attr = 10

# Output: (5, 0, 10)
print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
print(num1.attr)
```

Output

```
2+3j
(5, 0, 10)
Traceback (most recent call last):
  File "<string>", line 27, in <module>
    print(num1.attr)
AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

In the above example, we defined a new class to represent complex numbers. It has two functions, `__init__()` to initialize the variables (defaults to zero) and `get_data()` to display the number properly.
An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute `attr` for object `num2` and read it as well. But this does not create that attribute for object `num1`.

Deleting Attributes and Objects

Any attribute of an object can be deleted anytime, using the `del` statement. Try the following on the Python shell to see the output.

```
>>> num1 = ComplexNumber(2,3)
>>> del num1.imag
>>> num1.get_data()
Traceback (most recent call last):
```

```
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.get_data
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'get_data'
```

We can even delete the object itself, using the del statement.

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

Actually, it is more complicated than that. When we do `c1 = ComplexNumber(1,3)`, a new instance object is created in memory and the name `c1` binds with it.
On the command `del c1`, this binding is removed and the name `c1` is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.
This automatic destruction of unreferenced objects in Python is also called garbage collection.

2. Python Inheritance

Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more. In this tutorial, you will learn to use inheritance in Python.

Inheritance in Python

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**.

Python Inheritance Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example. A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
```

```
        print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides `n` and magnitude of each side as a list called `sides`.
The `inputSides()` method takes in the magnitude of each side and `dispSides()` displays these side lengths.
A triangle is a polygon with 3 sides. So, we can create a class called `Triangle` which inherits from `Polygon`. This makes all the attributes of `Polygon` class available to the `Triangle` class.
We don't need to define them again (code reusability). `Triangle` can be defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class `Triangle` has a new method `findArea()` to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

We can see that even though we did not define methods like inputSides() or dispSides() for class Triangle separately, we were able to use them.
If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Method Overriding in Python

In the above example, notice that __init__() method was defined in both classes, Triangle as well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, __init__() in Triangle gets preference over the __init__ in Polygon.
Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Polygon.__init__() from __init__() in Triangle).
A better option would be to use the built-in function super().
So, super().__init__(3) is equivalent to Polygon.__init__(self,3) and is preferred. To learn more about the super() function in Python, visit Python super() function.
Two built-in functions isinstance() and issubclass() are used to check inheritances.
The function isinstance() returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False
```

```
>>> isinstance(t,object)
True
```

Similarly, issubclass() is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

3. Python Multiple Inheritance

Python Multiple Inheritance

A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.
In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

Example

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

Here, the `MultiDerived` class is derived from `Base1` and `Base2` classes.

Python Multilevel Inheritance

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class. An example with corresponding visualization is given below.

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

Method Resolution Order in Python

Every class in Python is derived from the `object` class. It is the most base type in Python.
So technically, all other classes, either built-in or user-defined, are derived classes and all objects are instances of the `object` class.

```
# Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))

# Output: True
print(isinstance("Hello",object))
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice.

So, in the above example of `MultiDerived` class the search order is [`MultiDerived`, `Base1`, `Base2`, `object`]. This order is also called linearization of `MultiDerived` class and the set of rules used to find this order is called **Method Resolution Order (MRO)**.
MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents. In case of multiple parents, the order is the same as tuples of base classes.

MRO of a class can be viewed as the `__mro__` attribute or the `mro()` method. The former returns a tuple while the latter returns a list.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
<class '__main__.Base1'>,
<class '__main__.Base2'>,
<class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
<class '__main__.Base1'>,
<class '__main__.Base2'>,
<class 'object'>]
```

Here is a little more complex multiple inheritance example and its visualization along with the MRO.

```
# Demonstration of MRO

class X:
    pass


class Y:
    pass
```

```
class Z:
    pass


class A(X, Y):
    pass


class B(Y, Z):
    pass


class M(B, A, Z):
    pass

# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
#  <class '__main__.A'>, <class '__main__.X'>,
#  <class '__main__.Y'>, <class '__main__.Z'>,
#  <class 'object'>]

print(M.mro())
```

Output

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.X'>,
<class '__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```

4. Python Operator Overloading

You can change the meaning of an operator in Python depending upon the operands used. In this tutorial, you will learn how to use operator overloading in Python Object Oriented Programming.

Python Operator Overloading

Python operators work for built-in classes. But the same operator behaves differently with different 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.

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

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


p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output

```
Traceback (most recent call last):
  File "<string>", line 9, in <module>
    print(p1+p2)
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Here, we can see that a `TypeError` was raised, since Python didn't know how to add two `Point` objects together.
However, we can achieve this task in Python through operator overloading. But first, let's get a notion about special functions.

Python Special Functions

Class functions that begin with double underscore `__` are called special functions in Python.
These functions are not the typical functions that we define for a class. The `__init__()` function we defined above is one of them. It gets called every time we create a new object of that class.
There are numerous other special functions in Python. Visit Python Special Functions to learn more about them.

Using special functions, we can make our class compatible with built-in functions.

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

Suppose we want the `print()` function to print the coordinates of the `Point` object instead of what we got. We can define a `__str__()` method in our class that controls how the object gets printed. Let's look at how we can achieve this:

```
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)
```

Now let's try the `print()` function again.

```
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)


p1 = Point(2, 3)
print(p1)
```

Output

```
(2, 3)
```

That's better. Turns out, that this same method is invoked when we use the

built-in function `str()` or `format()`.

```
>>> str(p1)
'(2,3)'

>>> format(p1)
'(2,3)'
```

So, when you use `str(p1)` or `format(p1)`, Python internally calls the `p1.__str__()` method. Hence the name, special functions.
Now let's go back to operator overloading.

Overloading the + Operator

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 a `Point` object of the coordinate sum.

```
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)
```

Now let's try the addition operation again:

```
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)
```

What actually happens is that, when you 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.
Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`

Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Suppose we wanted to implement the less than symbol `<` symbol in our `Point` class.
Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
# overloading the less than operator
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 __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)
```

```
# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)
```

Output

```
True
False
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2
Less than or equal to	p1 <= p2	p1.__le__(p2
Equal to	p1 == p2	p1.__eq__(p
Not equal to	p1 != p2	p1.__ne__(p
Greater than	p1 > p2	p1.__gt__(p2
Greater than or equal to	p1 >= p2	p1.__ge__(p

Chapter 4: Python Advance Constructs

1. Python Modules

What are Modules in Python?

Consider a book store, in the book store there, is a lot of section like fiction books, physics books, financial books, language books and a lot more, and in each section, there are 100s of books present in just one section.

From the above example, consider the book store a folder, and section are considered to python files, and last the books in each section are called python functions, classes or python variables.

Formal Definition of Python Module

Python module can be defined as a python file that contains python definitions and statements. It contains python code along with python functions, classes, or python variables. From the above example, we can consider that each section is a python module.

In order words, we can say that the python module is a file with the .py extension.

Features of Python Module

- Modules provide us the flexibility to organize the code logically.
- Modules help to break down large programs into small manageable and organized files.
- It provides reusability of code.

Example:

Let's create a small module.

Create a file named example.py. The name of the file is example.py but the name of the module is example.

Now, in the example.py file, we will create a function called print_name().

```
# We are in the example module

def print_name(name):
```

```
    """
    This function does not return anything.
    It will just print your name.
    """
    print("Hello! {}, You are in example module.".format(name))
```

Now, we have created a python module example.

Now, let's try whether it works for us or not.

Now, create a second file at the same location where we have created our example module. Let's name that file as test.py.

Input:

```
import example

name = "scaler academy"
example.print_name(name)
```

Output:

```
Hello! scaler academy, You are in the example module.
```

Types of Python Modules

There are 2 types of python modules:

1. InBuilt modules
2. User-Defined Modules

InBuilt Module:

There are several modules built-in modules available in python.

Built-In modules like: **math sys os random numpy** and more.

How to call a Built-In module?**

Input:

```
#calling modules
import math # this is math module
import random # this is a random module


# now we use some of the math and random module function to check whether the modules are working or not?
cos30 = math.cos(30)
tan10 = math.tan(10)
pie = math.pi

# now using random module to generate some random numbers
random_int = random.randint(0,20)
```

```
print(f"Value of cos30 is: {cos30}")
print(f"Value of tan10 is: {tan10}")
print(f"Value of pie is: {pie}")
print(f"The random number generated using random int function: {random_int}")
```

Output:

```
Value of cos30 is: 0.15425144988758405
Value of tan10 is: 0.6483608274590866
Value of pie is: 3.141592653589793
The random number generated using random int function: 12
```

random.randint() function will generate a new number whenever the function is called. It will generate a new number in the given range. i.e., [0, 20)

Variables in Module

Already discussed that modules contain functions, classes. But apart from function and classes, modules in python also contain variables in them. Variables like tuples, lists, dictionaries, objects, etc.

Example:

Let's create one more module variables, and saved the .py as variables.py

```
# This is variables module
## this is a function in module
def factorial(n):
    if n == 1 || n == 0:
        return 1
    else:
        return n * factorial(n-1)

## this is dictionary in module
power = {
    1 : 1,
    2 : 4,
    3 : 9,
    4 : 16,
    5 : 25,
    6 : 36,
    7 : 49,
    8 : 64,
    9 : 81,
    10 : 100
}
```

```
## this is list in module
alphabets = [a,b,c,d,e,f,g,h]
```

Create another python file, and use the variables module, to print the factorial of a number, print the power of 6, and what is the second alphabet in the alphabet_list.

```
import variables

fact_of_6 = variables.factorial(6)

power_of_6 = variables.power[6]

alphabet_2 = variables.alphabets[1]

print(f"The factorial of 6 is : {fact_of_6}")
print(f"Power of 6 is : {power_of_6}")
print(f"Second Alphabet is : {alphabrt_2}")
```

Output:

```
The factorial of 6 is 720
Power of 6 is 36
The second alphabet is b
```

The Import Statement

Till now, we have created 2 modules i.e., example and variables. For using these 2 modules, we have 2 separate python to use the modules.

In both the files, we have used the import statement to use our modules example and variables.

What is an Import Statement?

The import statement is used to import all the functionality of one module to another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

How to Use Import Modules

For importing modules in python we need an import statement to import the modules in a python file.

Example:

```
import math

print(f"The value of pie using math module is : {math.pi}")
print("The value of pi, we studied is 3.14")
```

Output:

```
The value of pie using the math module is: 3.141592653589793
The value of pi, we studied is 3.14
```

Using from <module_name> import statement

When we are using the import statement to import either built-in modules or user-defined modules, that states we are importing all the functions/classes/variables available in that modules. But if we want to import a specific function/class/variable of that module.

We can import that specific function/class/variable, by using from <module_name> import statement we can import that specific resource of that module.

Syntax:

The syntax of from <module_name> import statement is:

```
from <module_name> import <function_name>, <class_name>, <function_name2>,
<variable_name>
```

Tip:

If we use '*' in place of function_name/class_name, it will import all the resources available in that module

```
from <module_name> import *
# this above statement will import all the available resources in the module.
```

Example:

Example 1

Input:

```
from math import *

print(pi)
print(sin(155))
print(tan(0))
```

Output:

```
3.141592653589793
-0.8733119827746476
0.0
```

Example 2

Input:

```
from math import sqrt, factorial, pi

print(sqrt(100))
print(factorial(10))
print(pi)
```

Output:

```
10.0
3628800
3.141592653589793
```

Python Module Search Path

In the previous sections, we have seen how to import the modules using an import statement, but how the python search for the path of the module.

In python, when we use the import statement to import the modules. The python interpreter looks for several locations.

After using the import statement to import modules. First, python interpreter will check for the built-in modules. If the module is not found, then python interpreter will look for the directories that are defined in the path sys. path.

The order of search is in below order:

- At first, the interpreter looks in the current working directory.
- If the module is not found in the current working directory, then the interpreter will search each directory in the PYTHONPATH environmental variable.
- If now also the module is not found, then it will search in the installation default directory.

```
import sys
print(sys.path)
```

Naming a Module

In the last 2 examples, we have created 2 python modules variables and example, and the modules files as variables.py and example.py. So, in this case, we get our modules names

as their file names.

So, overall we can say that we will get the name of a module by saving our python file.

Re-Naming a Module

Python provides an easy and flexible way to rename our module with a specific name, and then use that name to call our module resources.

For re-naming a module, we use the as keyword. The syntax is as follow:

```
import math as m
import numpy as np
import random as r

print(m.pi)
print(r.randint(0,10))
print(np.__version__)
```

Output

```
3.141592653589793
10
1.22.0
```

Reloading a Module

We can import a module only once in a session.

Suppose you using 2 modules variables and example at the same in a python file, and the variables module is updated while you were using these modules, and you want the updated code of that module which is updated. Now, we know that only one time we can import a module in our session.

So, we will use the reload function which is available in the imp module to get the updated version of our module.

```
import variables
import imp
imp.reload(variables)
<module 'variables' from '.\\variables.py'>
```

The dir() built-in Function

The dir() built-in function returns a sorted list of strings containing the names

defined by a module. The list contains the names of all the modules, variables, and functions that are defined in a module.

```
import math

print(dir(math))
```

Output:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin'
```

2. Packages in Python and Import Statement

Introduction

The meaning of Packages lies in the word itself. Packages do the work of organizing files in Python. Physically a package is a folder containing sub-packages or one or more modules (or files). They provide a well-defined organizational hierarchy of modules in Python. Packages usually are named such that their usage is apparent to the user. Python packages ensure modularity by dividing the packages into sub-packages, making the project easier to manage and conceptually clear.

To better understand what Packages in Python mean, let’s take an example from real life. Your family is shifting to a new apartment. Your mom has asked you to pack your books, CDs and toys so that it’s easy to unpack and organize them in your new house. The simplest way to do this would be to pack these items separately into three unique packages (boxes for CDs, books and toys, respectively) and name them based on their utility. Furthermore, you can also add sections to your book package based on the genre of books. We can do the same thing for CDs and toys as well.

Have you come across this way of organization somewhere? The most common example would be that of files on your phone. All your files would be saved in a folder that could further be distinguished based on the source

(WhatsApp Documents, Downloads, etc.).

Every package in Python must contain an __init__.py file since it identifies the folder as a package. It generally contains the initialization code but may also be left empty. A possible hierarchical representation of the shifting items package in the first example can be given below. Here, shifting items is our main package. We further divide this into three sub packages: books, CDs and toys. Each package and subpackage contains the mandatory __init__.py file. Further, the sub packages contain files based on their utility. Books contain two files fiction.py and non_fiction.py, the package CDs contain music.py and dvd.py, and the package toys contain soft_toys.py and games.py.

Importing Module From a Package in Python

Packages help in ensuring the reusability of code. To access any module or file from a Python package, use an import statement in the Python source file where you want to access it. Import of modules from Python Packages is done using the dot operator (.). Importing modules and packages helps us make use of existing functions and code that can speed up our work.

Syntax:

```
import module1[, module2,... moduleN]
```

Where import is a keyword used to import the module, module1 is the module to be imported. The other modules enclosed in brackets are optional and can be mentioned only when more than 1 module is to be imported.

Consider the writing package given below.

To use the edit module from the writing package in a new file test.py that you have created, you write the following code in the test.py:

```
import Writing.Book.edit
```

To access a function called plagiarism_check() of the edit module, you use the following code:

```
Writing.Book.edit.plagiarism_check()
```

The calling method seems lengthy and confusing, right? Another way to import a module would be to simply import the package prefix instead of the whole package and call the function required directly.

```
from Writing.Book import edit
plagiarism_check()
```

However, the above method may cause problems when 2 or more packages have similarly named functions, and the call may be ambiguous. Thus, this method is avoided in most cases.

Installing a Python Package Globally

To ensure system-wide use of the Python package just created, you need to run the setup script. This script uses the setup() function from the setuptools module. As a prerequisite, you need to have the latest versions of pip and setuptools installed on your system. Pip and setuptools are usually installed along with the Python binary installers. To upgrade the version of pip, use the following command:

```
python -m pip install --upgrade pip
```

Once you confirm that you have the latest versions of pip and setuptools installed, you can create a setup.py file in the main package (Writing) folder. The setup() function that will be imported here takes various arguments like the package's name, version, description, author, license, etc. The zip_safe argument is used to know the mode of storage of the package (compressed or uncompressed).

```
from setuptools import setup

setup(name='Writing',
version='1.0',
description='A Sample Package for all Writing Modules',
url='#',
author='username',
author_email='username@gmail.com',
license='MIT',
packages=['Writing'],
zip_safe=False)
```

Now to install the Writing package, open a terminal in your parent package folder (Writing) and type in the following command:

```
pip install Writing
```

The Writing package, which contains functions to assist writing (like count_words which is used to count the number of words in a string) will now be available for use anywhere on the system and can be imported using any script or interpreter by using the following commands:

```
import Writing
Writing.count_words(hello.txt)
```

This is how you create a system-wide package. If you want to create a package that can be used by users globally, follow the following steps:

1. Organize your package well with a readme.md, license and other files. Give an easy-to-understand and memorable name for your package.
2. Register an account on https://pypi.org.
3. Use twine upload dist/* to upload your package on PyPi and enter the credentials you used to create the account. The package will be uploaded directly to PyPi.
4. You can now install your Python package from PyPi using the command python -m pip install [package_name] in your terminal.

Import Statement in Python

The files in packages that contain python code are called modules. Modules are used to break down large code into smaller and more understandable parts. Commonly used code (functions) can be written in the form of modules and imported as and when needed, thus ensuring the reusability of code.

Let’s create a function prod(x,y) to multiply two numbers x and y, which are passed as arguments to a function and store it in a module named product.py.

```
def prod(x, y):
  res = x*y
  return res
```

To import the prod function in another module or use it in the interactive interpreter shell of Python, type:

```
import product
```

This statement does not import the module's functions into the symbol table. To access functions of the given module, we use the dot (.) operator as follows:

```
product.prod(3,5)
```

Output:

```
15
```

Python has a lot of in-built modules which can be accessed in the same way. For example, to print the value of pi, we can import the math module.

Example:

```
import math
print(math.pi)
```

Output:

```
3.141592653589793
```

Importing Attributes Using the from Import Statement in Python

We can also import individual attributes from the module. This reduces the complexity of writing the function calls.

Syntax:

```
from <module_name> import <attribute_name(s)>
```

In the previous example, we could import pi directly from the math module as follows:

```
from math import pi
print(pi)
```

Output:

```
3.141592653589793
```

You can import more than one attribute by separating them with commas (,). In the below example, we import the functions pi, floor (used to find the smallest integer greater than or equal to the given number) and fabs (used to find the absolute value of a number) from the math package.

```
from math import pi, floor, fabs
print(pi)
print(floor(3.55)
print(fabs(-2.5))
```

Output:

```
3.141592653589793
4
2.5
```

Importing Modules Using the From Import * Statement in Python

To import all modules' definitions, use the asterisk (*) operator.

Syntax:

```
from <module_name> import *
```

Example:

```
from math import *
print(pi)
```

Output:

```
3.141592653589793
```

Importing Modules in Python Using the Import as Statement

You can alter how a module is called in your program by aliasing it, i.e. giving it another name. The as operator is used for this purpose.

Syntax:

```
import <module_name> as <new_module_name>
```

Example:

```
import math as m
print(m.pi)
```

Output:

```
3.141592653589793
```

Importing Class/Functions from Module in Python

To import classes or functions from a module in Python, use the following syntax:

```
from <module_name> import <class_name/function_name>
```

Example: To import the prod function from the math module, which is used to calculate the product of 2 given numbers, we import prod specifically as shown. We then can access prod() directly without using the dot (.) operator.

```
from math import prod
print(prod(5,15))
```

Output:

```
75
```

Import User-defined Module in Python

To import modules defined by the user from different code files, use the following syntax:

```
import <user_defined_module_name>
```

Example: Consider the file example.py having a function val(x,y) defined as follows:

```
def val(x, y):
  res = x+y*5
```

```
 return res
```

To import the val function in another module or use it in the interactive interpreter shell of Python, type:

```
import example
```

This statement does not import the module's functions into the symbol table (a table in python generated by the compiler used to calculate the scope of identifiers). To access functions of the given module, we use the dot (.) operator as follows:

```
example.val(3,5)
```

Output:

```
28
```

Importing from Another Directory in Python

Usually, when we import modules, the called modules are placed in the same directory as the calling modules. To import modules from another directory (collection of files in a path different from the present working directory), we use the importlib library by calling the following import statement.

```
import importlib
```

Methods like exec_module (used to execute a module)
and module_from_spec (used to create a new module from the given specifications) are used to satisfy the purpose.

Example: Consider two modules: upperCase.py (which is to be imported), and final.py (the one that imports upperCase.py). The directory structure is as follows:

- PythonCode
- upperCase.py
- MainCodes
- final.py

You will realize that upperCase.py and MainCodes are in a common directory named PythonCode. final.py is placed in the MainCodes subdirectory. Thus, upperCase.py and final.py do not share the same directory.

The modules upperCase.py and final.py are defined as follows:

upperCase.py - This contains a function called upperCase which converts the string sent as an argument to upperCase and returns the new string. This is done by first converting the argument to a string (to ensure that

no ambiguity in the type of argument remains) and then using the upper() method to fulfil the required purpose.

```
def upperCase(x):
  return str(x).upper()
```

final.py - This is the file where the function upperCase needs to be imported. Since upperCase lies in another directory (as shown in the directory structure above), we import
the importlib and importlib.util packages. A function called module_directory contains two parameters: name_module and path representing the module's name and the module's path to be imported, respectively. A variable P is used to save the ModuleSpec, i.e. all the import-related information required to load the module. It is done by calling the spec_from_file method
from importlib.util after passing the name_module and path as parameters. Another variable import_module saves the module from the ModuleSpec saved in the variable P. The exec_module is used to execute the import_module module in its namespace. The import_module is then returned by the function. A result variable is used to call the module_directory function. The upperCase function is then called using a dot (.) operator.

```
import importlib, importlib.util
def module_directory(name_module, path):
  P = importlib.util.spec_from_file_location(name_module, path)
  import_module = importlib.util.module_from_spec(P)
  P.loader.exec_module(import_module)
  return import_module
result = module_directory("result", "../upperCase.py")
print(result.upperCase('SaFa'))
```

Output:

```
SAFA
```

Importing Class from Another File in Python

We just saw how to import functions from files in other directories. What if you want to import a class from another file? The importlib library is used to import classes from another file, which can be saved in the same directory or another.

Example: Consider 2 files: calc.py (which is to be imported), and final.py (the one that imports calc.py). The directory structure is as follows: PythonCode calc.py MainCodes final.py

You will realize that calc.py and MainCodes are in a common directory named PythonCode. final.py is placed in the MainCodes subdirectory. Thus, calc.py and final.py do not share the same directory.

The modules calc.py and final.py are defined as follows:

calc.py - This contains a class called calc which has 3 user-defined class functions namely add(), sub() and mul() used to implement addition, subtraction and multiplication of 2 numbers respectively. In addition to that, calc.py also contains a function named greet() which when called, prints 'Hello World!' on the screen.

```
class calc:
  def add(self, x, y):
     return x + y
  def sub(self, x, y):
     return x - y
  def mul(self, x, y):
     return x * y

def greet():
  print("Hello World!")
```

final.py - This is the file where the class calc needs to be imported. Since calc lies in another directory (as shown in the directory structure above), we import the importlib and importlib.util packages. A function called module_directory contains two parameters: name_module and path that represent the module's name and the module's path to be imported respectively. A variable P is used to save the ModuleSpec, i.e. all the import-related information required to load the module. This is done by calling the spec_from_file method from importlib.util after passing the name_module and path as parameters. Another variable import_module saves the module from the ModuleSpec saved in the variable P. The exec_module is used to execute the import_module module in its own namespace. The import_module is then returned by the function. A result variable is used to call the module_directory function. An object named object is then created by instantiating the class calc from the imported module. Different functions of the calc class can be then called by using the dot (.) operator as shown in the code below.

```
import importlib, importlib.util
def module_directory(name_module, path):
  P = importlib.util.spec_from_file_location(name_module, path)
import_module = importlib.util.module_from_spec(P)
```

```
  P.loader.exec_module(import_module)
  return import_module
result = module_directory("result", "../calc.py")
# Created a class object
object = result.calc()

# Calling and printing class methods
print(object.add(20,5))
print(object.sub(20,5))
print(object.mul(20,5))

# Calling the explicitly created function
result.greet()
```

Output:

```
25
15
100
Hello World!
```

3. Python Collection Module

Introduction to Modules

The Modules in Python are simply the ".py" or python files that contain Python code that can contain code for specific inbuilt function and can be imported into some other Python program to make use of inbuilt functions easier.

A module can be thought as a code library or a file containing a group of methods and functions that you want to include in your python program. We can use modules to group together related functions, classes, or code blocks in the same file. As a result, splitting large Python code blocks into modules comprising code is regarded a best practise for building larger Python code.

Collection Module

The collection Module in Python has various types of containers which are available to use easily by importing them into your program using the collection module.

A Container is a type of object that can be used to hold multiple items while simultaneously providing a way to access and iterate over them, such as a Tuple or a list.

There are different containers which comes under collections module:

- namedtuple()
- OrderedDict()
- defaultdict()
- Counter()
- deque()
- Chainmap

Now lets talk about all these Containers in detail.

namedtuple()

The collections module's namedtuple() function is used to return a tuple with names for each of the tuple's positions. Each value of the tuple can be called using their corresponding names. Regular tuples have a difficult time remembering the index of each field of a tuple object. The namedtuple was created to address this problem, and it solves the index problem for regular tuple objects.

for importing and using the namedtuple() we use the below syntax:

Syntax:

```
from collections import namedtuple


name = namedtuple('name', 'x, y')
```

After importing, you can simply just initialize the object and can use it according to the code.

namedTuple function takes the object name as its first parameter followed by the names as its positions. First define the structure of namedtuple() object using the namedtuple() function with names as its positions and initialize the tuple object by passing the values to the initialzed object.

Lets see some examples related to namedtuple() function.

Example 1: First we will define the structure of namedtuple() object using the namedtuple() function with names as its positions.

We will initialize the tuple object by passing the values to the initialzed object.

Now you can use the initialised object to get the values by using the names as

the tuple’s positions.

```
# Importing the namedtuple
from collections import namedtuple

# Defining the name of the object and positions for the object
nameTup = namedtuple('nameTup', 'value1, value2, value3')
myTuple = nameTup('Scaler', 'Topics', 'Python')

# Printing the values
print(myTuple.value1)
print(myTuple.value2)
print(myTuple.value3)
```

Output: The program will print the values from the tuple object by using the names as their positions

```
Scaler
Topics
Python
```

Example 2: In this example, we will make use of a list to create the tuple and we will access the values both by using indexes and the names.

```
# Importing the namedtuple
from collections import namedtuple

# Defining the positions for the object
tuple1 = namedtuple('tuple1', ['name', 'address', 'profession'])
myTuple = tuple1('ABC', 'Street No.1 ABC complex', 'Teacher')


# Printing the values using index of the list
print('Using the indexes to get the values')

print('\nName is :', myTuple[0])
print('Address is :', myTuple[1])
print('Profession is :', myTuple[2])

# Printing the values using names as positions
print('\nUsing the names to get the values')

print('\nName is :', myTuple.name)
print('Address is :', myTuple.address)
print('Profession is :', myTuple.profession)

print('\nSimply printing the tuple')

print('\n',myTuple)
```

using a list for creating the tuple hepls us to access the values of the tuple by both using indexes and the names.

Output: The program will print the values from the tuple object by using the indexes of the list passed and also by the names as their positions.

Using the indexes to get the values
Name is: ABC
Address is: Street No.1 ABC complex
Profession is: Teacher

Using the names to get the values
Name is: ABC
Address is: Street No.1 ABC complex
Profession is: Teacher

Simply printing the tuple
tuple1(name='ABC', address='Street No.1 ABC complex', profession='Teacher')

OrderedDict()

OrderedDict is a dictionary in which the keys are always inserted in the same order. The order of the keys in a dictionary is preserved if they are inserted in a specific order. Even if the value of the key is changed later, the position will not change.

So if we will iterate over the Dictionary it will always print out the values in the order they are inserted.

for importing the OrderedDict() we use the below syntax:

Syntax:

```
from collections import OrderedDict

myDict = OrderedDict()
```

After importing, you can simply just initialize the object and can use it according to the code.

First define the structure using the OrderedDict() function and initialize the dictionary object by passing the values according to the keys.

Further you can use inbuilt function for dictionary like:

- d.clear()
- d.get([,])
- d.items()
- d.keys()
- d.values()
- d.pop([,])
- d.popitem()
- d.update()

Lets see some examples related to OrderedDict() function.

Example 1:

First we will define the structure using the OrderedDict() function and we will initialize the dictionary object by passing the values according to the keys.

Now you can use the dictionary to get the key-values.

```
# Importing the dictionary
from collections import OrderedDict

# Initializing the dictionary
myDict = OrderedDict()

# Adding the values
myDict['key1'] = 'value1'
myDict['key2'] = 'value2'
myDict['key3'] = 'value3'
myDict['key4'] = 'value4'

# Printing the dictionary
print(myDict)
```

We have simply initiated 4 keys and values in the ordered dictionary.

Output:

You can see that the order is preserved when we print the dictionary.

```
OrderedDict([('key1', 'value1'), ('key2', 'value2'), ('key3', 'value3'), ('key4', 'value4')])
```

Example 2:

In this example, we will make an OrderedDict and then we will change the key values but the original order will be maintained.

```
# Importing the dictionary
from collections import OrderedDict

# Initializing the dictionary
myDict = OrderedDict()

# Adding the values
myDict['A'] = '1'
myDict['B'] = '2'
myDict['C'] = '3'
myDict['D'] = '4'

# Changing the values
myDict['A'] = '5'

# Ordering will be same
```

```
print(myDict)
```

Changing the value of key "A" from 1 to 5 will not affect the ordering of the keys in the dictionary.

Output:

You can see that the order is preserved even after changing the key-value of the dictionary.

OrderedDict([('A', '5'), ('B', '2'), ('C', '3'), ('D', '4')])

defaultdict()

In Python, a dictionary is an unordered collection of data values that can be used to store data values in the same way that a map can. The Dictionary has a key-value pair and the key must be unique and immutable and it throws error when we try to access non-existent key. The only difference between defaultdict() and simple Dictionary is that when you try to access a non-existent key in defaultdict(), it does not throw an exception or a key error.

For importing defaultdict() we use the below syntax:

Syntax:

```
from collections import defaultdict

mydict = defaultdict()
```

After importing, you can simply just initialize the object and can use it according to the code.

First define the structure using the defaultdict() function and initialize the dictionary object by passing the values according to the keys.

Further you can use inbuilt function for dictionary like:

- d.clear()
- d.get([,])
- d.items()
- d.keys()
- d.values()
- d.pop([,])
- d.popitem()
- d.update()

Lets see some examples related to defaultdict() function.

Example 1:

First we will define the structure using the defaultdict() function and we will

initialize the dictionary object by passing the values according to the keys.

Now you can use the dictionary to get the key-values.

```
# Importing the dictionary
from collections import defaultdict

# Initializing the dictionary
defDict = defaultdict()

# Adding the values
defDict['k1'] = 'v1'
defDict['k2'] = 'v2'
defDict['k3'] = 'v3'
defDict['k4'] = 'v4'

# Printing the dictionary
print(defDict)
```

We have simply initiated 4 keys and values in the ordered dictionary.

Output:

This example simply prints the dictionary values.

defaultdict(None, {'k1': 'v1', 'k2': 'v2', 'k3': 'v3', 'k4': 'v4'})

Example 2:

In this example, we will try to access the key which is not present in the dictionary but the defaultdict will not give any key error.

```
# Importing the dictionary
from collections import defaultdict

# Initializing the dictionary
dic = defaultdict(int)

# Adding the values
dic[1] = 'a'
dic[2] = 'b'
dic[3] = 'c'

# Printing the dictionary
print(dic[4])
```

defaultdict is advantageous in the conditions where inexisting keys are accessed as error does not occurs in those cases.

Output:

While initializing the defaultdict, if we don't pass an object reference such as int, bool, it will result in a key error and won't be able to deduce the fallback default value in such scenarios.

In this case, we have set default value as int so in case of missing key it will return 0.

You can see that when we tried to access the value 4 in the dictionary, it just returned 0 instead of returning KeyError.

0

Counter()

A counter is a built-in data structure that counts the number of times each value in an array or list appears.

Hashable objects are counted using the Counter subclass. counter() object constructs an iterable hash table when called.

For importing Counter() we use the below syntax: **Syntax:**

```
from collections import Counter
```

After importing, you can simply just initialize the Counter object and can use it according to the code. Lets see some examples related to Counter() function.

Example 1:

First we will initialize the structure using the Counter() function and we will pass the list in the counter object and a hashable object will be created

Now you can get the key-values of the hash.

```
# Importing the Counter
from collections import Counter

# Making a list
l = [1,1,1,1,1,2,2,3,3,4]

# Making the counter Object
c = Counter(l)
print(c)
```

Counter object makes a dictionary of all distinct keys and their counter values.

Output:

This example simply prints the counter object created. **Counter({1: 5, 2: 2, 3: 2, 4: 1})**

Example 2:

we will pass the list containing the string values and hence the dictionary of string as keys will be created.

Now you can get the key-values of the hash.

```
# Importing the Counter
from collections import Counter

# Making a list
stringList = ['a', 'b', 'c', 'c', 'c', 'b', 'b', 'b', 'a', 'a', 'a', 'a']

# Making the counter Object
c = Counter(stringList)
print(c)
```

Output:

This example simply prints the counter object created. **Counter({'a': 5, 'b': 4, 'c': 3})**

deque()

Deque is a more efficient variant of a list for adding and removing items easily. It can add or remove items from the beginning or the end of the list.

It is a double ended queue which has insertion and deletion operations available at both the ends.

For importing deque() we use the below syntax:

Syntax:

```
from collections import deque

deq = deque()
```

After importing, you can simply just initialize the deque() object and can use it according to the code.

After importing, you can simply just initialize the object and pass the list in the deque object.

Further you can use inbuilt function for deque like:

- append(): append to right of deque()
- appendleft(): append to left of deque()
- pop(): pop an element from the right of the deque()
- popleft(): pop for the left of deque()
- index(ele, beg, end): get the index of the element from deque()
- insert(i, a): insert at a particular index of the deque()
- remove(): remove an element from deque
- count(): count occurences of an element in deque()
- reverse(): reverse the deque()
- rotate(): rotate the deque() by specified number. Lets see some examples related to deque().

Example:

First we will initialize the deque object and we will pass the list in the deque object.

Now we will be applying the removal and insertion on both the ends of the deque.

We added the items at both ends using append() and appendleft() function and then printed it.

Now finally we have removed the items from both the ends using pop() and popleft() function.

```
# Importing the deque
from collections import deque

# Initialization
l = ['Hi', 'This', 'is', 'Scaler']
myDeq = deque(l)

# Printing the deque
print("Original Deque is :", myDeq)

# Inserting at both the ends
myDeq.append("!!")
myDeq.appendleft("!!")

# Printing the updated deque
print("Deque after adding to both the ends is :",myDeq)

# Removing from both the ends
myDeq.pop()
print("Deque after removal from start is :",myDeq)

myDeq.popleft()
print("Deque after removal from beginning is :",myDeq)

myDeq.insert(1, "New")
print("Deque after insertion is :",myDeq)

myDeq.remove("New")
print("Deque after removal of value New is :",myDeq)

print("count of Scaler in deque is : ", myDeq.count("Scaler"))

myDeq.reverse()
print("Deque after reversing is :",myDeq)

myDeq.rotate(1)
print("Deque after rotation by 1 element is :",myDeq)
```

We have performed every above mentioned functions on our deque.

Output:

First simply a list is passed in the deque() function, Then we added the items at both ends using append() and appendleft() function and then printed it.

Now finally we have removed the items from both the ends using pop() and popleft() function.

Original Deque is: deque(['Hi', 'This', 'is', 'Scaler'])
Deque after adding to both the ends is: deque(['!!', 'Hi', 'This', 'is', 'Scaler', '!!'])
Deque after removal from start is: deque(['!!', 'Hi', 'This', 'is', 'Scaler'])**
Deque after removal from beginning is: deque(['Hi', 'This', 'is', 'Scaler'])
Deque after insertion is: deque(['Hi', 'New', 'This', 'is', 'Scaler'])
Deque after removal of value New is: deque(['Hi', 'This', 'is', 'Scaler'])
count of Scaler in deque is: 1
Deque after reversing is: deque(['Scaler', 'is', 'This', 'Hi'])
Deque after rotation by 1 element is: deque(['Hi', 'Scaler', 'is', 'This'])

Chainmap

ChainMap returns a list of dictionaries after combining them and chaining them together. ChainMaps can encapsulate a multiple dictionaries into a single object or unit and has no limitations for the number of dictionaries to contain.

For importing Chainmap() we use the below syntax:

Syntax:

```
from collections import Chainmap

cm = chainmap(dic1, dic2)
```

After importing, you can simply just initialize the Chainmap() object by passing the dictionaries you want to encapsulate.

Further you can use inbuilt function for chainmap like:

- keys(): it is used to get all the keys of chainmap
- values(): it is used to get all the values of chainmap
- maps(): it is used to get all the keys and their particular values in the object
- newchild(): It adds a new dictionary at the beginning of the chainmap
- reversed(): It is used to reverse the chainmap object.

Lets see some examples related to Chainmap().

Example:

First we will initialize the Chainmap object and we will pass all the dictionaries that we wanna encapsulate in the chainmap object.

Finally we will print the final chainmap object.

```
# Importing the deque
from collections import ChainMap

# Dictionary 1
dict1 = { 'k1' : 1, 'k2' : 2, 'k3': 3 }
# Dictionary 2
dict2 = { 'k4' : 4 }

# Making the ChainMap
finalChainMap = ChainMap(dict1, dict2)

# Printing the ChainMap
print(finalChainMap)

print("All the keys of the chainmap is :", list(finalChainMap.keys()))

print("All the values the chainmap is :", list(finalChainMap.values()))

print("All the keys-value pairs of the chainmap is :", finalChainMap.maps)

print("All the keys of the chainmap is :", list(finalChainMap.keys()))

dict3 = { 'k5' : 5 }

chain1 = finalChainMap.new_child(dict3)
print("chainmap after addition at beginning is :", chain1)

print("Reversed chainmap is :", list(reversed(chain1.maps)))
```

Output:

We Simply Just printed out the chainMap object after encapsulating the dictionaries to the Object.

ChainMap({'k1': 1, 'k2': 2, 'k3': 3}, {'k4': 4})

All the keys of the chainmap is: ['k2', 'k3', 'k1', 'k4']

All the values the chainmap is: [2, 3, 1, 4]

All the keys-value pairs of the chainmap is: [{'k1': 1, 'k2': 2, 'k3': 3}, {'k4': 4}]

All the keys of the chainmap is: ['k2', 'k3', 'k1', 'k4']

chainmap after addition at beginning is: ChainMap({'k5': 5}, {'k1': 1, 'k2': 2, 'k3': 3}, {'k4': 4})

```
Reversed chainmap is: [{'k4': 4}, {'k1': 1, 'k2': 2, 'k3': 3}, {'k5': 5}]
```

4. Regular Expression in Python

Regular expression is a sequence of characters that forms a pattern which is mainly used to find or replace patterns in a string. These are supported by many languages such as python, java, R etc.Most common uses of regular expressions are:

1. Finding patterns in a string or file.(Ex: find all the numbers present in a string)
2. Replace a part of the string with another string.
3. Search substring in string or file.
4. Split string into substrings.
5. Validate email format.

We will see examples of above mentioned uses in detail.

RegEx Module

In python we have a built-in package called re to work with regular expressions. We can use it as shown in the example below:

```
import re
text = "'Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London"'
res = re.search("^Alan.*London$",text)
if(res):
   print("We have a match!")
else:
   print("We don't have a match")


Output:
We have a match!
```

This will give “We have a match!” as output as the given string starts with Turing and ends with London. We will see how this works later in the article.

Python RegEx Expressions Functions

The ‘re’ package in python provides various functions to work with regular expressions. We will discuss some commonly used ones.

S No	Function	Description
1	findall(pattern,string)	This matches all the occurrences of the pattern present in t
2	search(pattern,string)	This matches the pattern which is present at any position in the s match the first occurrence of the pattern.
3	split(pattern,string)	This splits the string on the given pattern.
4	sub(pattern,rep_substring,string)	This replaces one or more matching pattern in the string with the

Meta Characters

These are the characters which have special meaning. The following are some of the meta characters with their uses.

S No	Meta character	Description
1	[ ](Square brackets)	This matches any single character in this bracket with the given string
2	. (Period)	This matches all the characters except the newline. If we pass this as a pattern in the fi will match with all the characters present in the string except newline chara
3	^ (Carret)	This matches the given pattern at the start of the string.This is used to check if the str particular pattern or not.
4	$ (Dollar)	This matches the given pattern at the end of string. This is used to check if the string en or not.
5	* (Star)	This matches 0 or more occurrences of the pattern to its left.
6	+ (Plus)	This matches 1 or more occurrences of the pattern to its left.
7	? (Question mark)	This matches 0 or 1 occurrence of the pattern to its left.
8	{ } (Braces)	This matches the specified number of occurrences of pattern present in the b
9	(Alternation)	This works like ‘or’ condition. In this we can give two or more patterns. If the string one of the given patterns this will give a match.
10	( ) (Group)	This is used to group various regular expressions together and then find a match i
11	\ (Backslash)	This is used to match special sequences or can be used as escape characters

Special Sequences in Python RegEx

S No	Sequence	Description
1	\A	This gives a match if the characters to the right of this are at the beginning of the string.
2	\b	This gives a match if the characters to the right are at the beginning of a word or the chara are at the end of a word in the given string.
3	\B	This gives a match if the characters to the right or left of \B are not present at the beginni word in the given string.
4	\d	This gives a match if the string contains a digit.
5	\D	This gives a match if the string contains only non digit characters.
6	\s	This gives a match if the string contains a white space character.
7	\S	This gives a match if the string contains only characters other than white space character.
8	\w	This gives a match if the string contains any character in a-z, A-Z, 0-9 and underscore(_).

9	\W	This gives a match if the string contains characters other than a-z, A-Z, 0-9 and underscor
10	\Z	This gives a match if the characters to the left of \Z are present at the end of the string.

Examples for each sequence are given below

```
import re
text = '''Alan Turing was born on 23 June 1912 in London.'''
# Example for \A
res = re.findall('\AAlan',text)
print("Result for \A = ", res)
print("-"*79)
# Example for \b
res = re.findall(r'\bLon',text)
print("Result for \\b = ", res)
print("-"*79)
# Example for \b
res = re.findall(r'ring\b',text)
print("Result for \\b = ", res)
print("-"*79)
# Example for \B
res = re.findall('\Bon',text)
print("Result for \B = ", res)
print("-"*79)
# Example for \d
res = re.findall('\d',text)
print("Result for \d = ", res)
print("-"*79)
# Example for \D
res = re.findall('\D',text)
print("Result for \D = ", res)
print("-"*79)
# Example for \s
res = re.findall('\s',text)
print("Result for \s = ", res)
print("-"*79)
# Example for \S
res = re.findall('\S',text)
print("Result for \S = ", res)
print("-"*79)
# Example for \w
res = re.findall('\w',text)
print("Result for \w = ", res)
print("-"*79)
# Example for \W
res = re.findall('\W',text)
print("Result for \W = ", res)
print("-"*79)
# Example for \Z
res = re.findall('London.\Z',text)
```

```
print("Result for \Z = ", res)

Output:
Result for \A =  ['Alan']
------------------------------------------------------------------------------
Result for \b =  ['Lon']
------------------------------------------------------------------------------
Result for \b =  ['ring']
------------------------------------------------------------------------------
Result for \B =  ['on', 'on']
------------------------------------------------------------------------------
Result for \d =  ['2', '3', '1', '9', '1', '2']
------------------------------------------------------------------------------
Result for \D =  ['A', 'l', 'a', 'n', ' ', 'T', 'u', 'r', 'i', 'n', 'g', ' ', 'w', 'a', 's', ' ', 'b', 'o', 'r', 'n', ' ', 'o', 'n', ' ', ' ', 'J',
'u', 'n', 'e', ' ', ' ', 'i', 'n', ' ', 'L', 'o', 'n', 'd', 'o', 'n', '.']
------------------------------------------------------------------------------
Result for \s =  [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
------------------------------------------------------------------------------
Result for \S =  ['A', 'l', 'a', 'n', 'T', 'u', 'r', 'i', 'n', 'g', 'w', 'a', 's', 'b', 'o', 'r', 'n', 'o', 'n', '2', '3', 'J', 'u', 'n', 'e',
'1', '9', '1', '2', 'i', 'n', 'L', 'o', 'n', 'd', 'o', 'n', '.']
------------------------------------------------------------------------------
Result for \w =  ['A', 'l', 'a', 'n', 'T', 'u', 'r', 'i', 'n', 'g', 'w', 'a', 's', 'b', 'o', 'r', 'n', 'o', 'n', '2', '3', 'J', 'u', 'n', 'e',
'1', '9', '1', '2', 'i', 'n', 'L', 'o', 'n', 'd', 'o', 'n']
------------------------------------------------------------------------------
Result for \W =  [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '.']
------------------------------------------------------------------------------
Result for \Z =  ['London.']
```

Sets

A set is a set of characters inside the square bracket which is treated as a pattern. Given below are some examples of set:

No	Set	Description
1	[abcd]	Gives a match if the string contains a,b,c or d.
2	[a-z]	Gives a match if the string contains any character from a to z.
3	[A-Z]	Gives a match if the string contains any character from A to Z.
4	[0-9]	Gives a match if string contains digits from 0 to 9
5	[a-zA-Z0-9]	Gives a match if any of the above conditions holds true.
6	[^a-zA-Z]	Gives a match if the string doesn’t contain any alphabet.
7	[%&$#@*]	Gives a match if the string contains any of these characters. When these characters are ir they are treated as normal characters.

findall(pattern, string)

This function is the same as search but it matches all the occurrences of the pattern in the given string and returns a list. The list contains the number of times it is present in the string.

Ex: The following example will make it clear.

```
import re
text = "'Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London"'
res = re.findall('Turing',text)
print("Result = {}".format(res))

Output:
Result = ['Turing']
```

In the output you can clearly see that the function finds a match for the pattern ‘Turing’’. It is advisable to use findall while searching for a pattern in a string as it covers both match and search functions.

search(pattern, string)

This is the same as match function but this function can search patterns irrespective of the position at which the pattern is present. The pattern can be present anywhere in the string. This function matches the first occurrence of the pattern.

Ex: The following example shows how to use the function

```
import re
text = "'Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London"'

res = re.search('Turing',text)
print("Result = {} and start,end position = {}".format(res,res.span()))

Output:
Result = <re.Match object; span=(5, 11), match='Turing'> and start,end position = (5, 11)
```

The function returns re.Match object if pattern if present in the string else returns None.

We can also get the start and end positions of matching pattern by calling span method on the re.Match object.

split(pattern, string)

This function splits a string on the given pattern. This returns the result as a list after splitting. The example given below will make it clear.

```
import re
text = "'Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London"'

res = re.split("a", text)
print("Result = {}".format(res))
```

```
Output:
Result = ['Al', 'n Turing w', 's ', ' pioneer of theoretic', 'l computer science ', 'nd ', 'rtifici', 'l intelligence. He w', 's born on 23 June 1912 in M', 'id', ' V', 'le, London']
```

sub(pattern, repl, string)

This function replaces a pattern with the given substring in a given string. In the example below we will replace the word ‘theoretical’ with ‘practical’.

```
import re
text = "'Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London"'

res = re.sub('theoretical','practical',text)
print("Result = {}".format(res))

Output:
Result = Alan Turing was a pioneer of practical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London
```

In the output we can see that the function replaced the pattern ‘theoretical’ with the given substring ‘practical’’. This function will replace all the patterns present in the string with the given substring.

Match Object

Whenever we call any regex method/function it searches the pattern in the string. If it finds a match then it returns a match object else return None. We will see how the match object looks like and how to access methods and properties of that object. Let’s search a pattern in a string and print the match object.

```
import re
text = "'Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London"'
# Searches the pattern in the string.
res = re.search('computer',text)
print("Match object = {}".format(res))

Output:
Match object = <re.Match object; span=(41, 49), match='computer'>
```

In the example above we can see that if a match happens then the re.Match object is returned. If there is no match then None will be returned.

Now we will see the attributes and properties of re.Match objects one by one. They are as follows:

- **match.group()**: This returns the part of the string where the match was there.
- **match.start()**: This returns the start position of the matching pattern in the string.
- **match.end()**: This returns the end position of the matching pattern in the string.
- **match.span()**: This returns a tuple which has start and end positions of matching pattern.
- **match.re**: This returns the pattern object used for matching.
- **match.string**: This returns the string given for matching.
- **Using r prefix before regex**: This is used to convert the pattern to raw string.This means any special character will be treated as normal character. Ex: \ character will not be treated as an escape character if we use r before the pattern.

Now we will see examples for each of the functions mentioned above.

```
import re
text = '''Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London'''
# Searches the pattern in the string.
res = re.search('computer',text)
print("Match object = {}".format(res))
print("--"*30)
print("group method output = ",res.group())
print("--"*30)
print("start method output = ",res.start())
print("--"*30)
print("end method output = ",res.end())
print("--"*30)
print("span method output = ",res.span())
print("--"*30)
print("re attribute output = ",res.re)
print("--"*30)
print("string attribute output = ",res.string)
print("--"*30)

# Example of using r as prefix.
# Searching for \\ in the following string
text = r'search \\ in this string'
# searching using r as prefix
res = re.search(r"\\",text)
print("With r as prefix = ",res)
```

```
Output:
Match object = <re.Match object; span=(41, 49), match='computer'>
----------------------------------------------------------
group method output =  computer
----------------------------------------------------------
start method output =  41
----------------------------------------------------------
end method output =  49
----------------------------------------------------------
span method output =  (41, 49)
----------------------------------------------------------
re attribute output =  re.compile('computer')
----------------------------------------------------------
string attribute output =  Alan Turing was a pioneer of theoretical computer science and artificial intelligence. He was born on 23 June 1912 in Maida Vale, London
----------------------------------------------------------
With r as prefix =  <re.Match object; span=(7, 8), match='\\'>
```

If there is no match then instead of re.Match object, it will return ‘none’.

The concepts that you have learned above can be used to verify email address, phone number, address or names of colleges, institutes etc.

5. Python Datetime

Introduction to Python Datetime

Ever wondered how does the banking system manages date and time, like how do they calculate the remaining tenure of a loan. Well, the answer lies in the functions of python datetime module of python. Read along to know more.

DateTime module is provided in Python to work with dates and times. In python DateTime, is an inbuilt module rather than being a primitive data type, We just have to import the module mentioned above to work with dates as date object.

There are several classes in the datetime python module of python which help to deal with the date & time. Moreover, there are so many functions of these classes from which we can extract the date and time.

How to Use Date and Datetime Class?

Firstly, you need to import the python datetime module of python using the following line of code, so that you can use its inbuilt classes to get current date & time etc.

```
import datetime
```

1. Now, we can use the date class of the datetime module. This class helps to convert the numeric date attributes to date format in the form of YYYY-MM-DD. This class accepts 3 attributes Year, Month & Day in the following order (Year, Month, Day).

Syntax

```
import datetime

    var1 = datetime.date(YYYY, MM, DD)

# This will convert the numeral date to the date object
```

Example

```
import datetime

    userdate = datetime.date(2021, 10, 20)

    print('userdate: ', userdate)
    print('type of userdate: ', type(userdate))
```

Output:

```
success
  userdate:  2021-10-20
  type of userdate:  <class 'datetime.date'>
```

Explanation of code

- In the above code we imported the python datetime library.
    - We then saved the date time object into the variable.
    - Then we print the variable and also the type of variable whose output is down below.
- Now, we can use the datetime python class of the datetime module.
    - This class helps to convert the numeric date & time attributes to date-time format in the form of YYYY-MM-DD hr:min:s:ms.

This class accepts 7 attributes Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds in the following order (Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds).

Syntax

```
import datetime
```

```
    var1 = datetime.datetime(YYYY, MM, DD, hr, min, s, ms)

    # This will convert the numeral date to the datetime object in the form of
      YYYY-MM-DD hr:min:s:ms.
```

Example

```
import datetime

  userdatetime = datetime.datetime(2019, 5, 10, 17, 30, 20, 154236)

  print('userdatetime: ', userdatetime)
  print('type of userdatetime: ', type(userdatetime))
```

Output:

```
success
  userdatetime:  2019-05-10 14:30:20.154236
  type of userdatetime:  <class 'datetime.datetime'>
```

Explanation of code

- In the above code we imported the datetime library.
- We then saved the date time object into the variable.
- Then we print the variable and also the type of variable whose output is down below.

Classes of DateTime Python Module

Date

Now, we can use the date class of the datetime module.

This class helps to convert the numeric date attributes to date format in the form of YYYY-MM-DD.

This class accepts 3 attributes Year, Month & Day in the following order (Year, Month, Day).

Syntax

```
import datetime

  var1 = datetime.date(YYYY, MM, DD)

  # This will convert the numeral date to the date object
```

Example

```
import datetime
```

```
userdate = datetime.date(2021, 9, 15)

print('userdate: ', userdate)
print('type of userdate: ', type(userdate))
```

Output

```
userdate:  2021-09-15
type of userdate:  <class 'datetime.date'>
```

Explanation of code

- In the above code we imported the datetime library.
- We then saved the date time object into the variable.
- Then we print the variable and also the type of variable whose output is down below.

As we can see that we have changed the numeral data to the datetime object and the same can be verified by the type of object printed below.

Time

Now, we can use the time class of the python datetime module.

Time class returns the local time of the area where user is present.

Time class accepts 5 attributes hour, minute, second, microsecond & fold in the following order (hour, minute, second, microsecond, fold).

All the above mention attributes are optional, but the initial value of all the attributes are 0.

Points to Remember About Time Class

- If we don't pass any attribute in time class then it will return the default time which is 00:00:00.
- By specifying particular attribute in the time class then it will set its value and give the time as per the user requirement.

Syntax

```
import datetime

# returns the time with all it's value as 0 like 00:00:00
var1 = datetime.time()

# returns the time with the value which are specified by the user in the attributes
var2 = datetime.time(hour=?, minute=?, second=?, microsecond=?)
```

Example

```
import datetime
```

```
time1 = datetime.time()
print('without passing any attribute: ', time1)

time2 = datetime.time(hour=12, minute=55, second=50)
print('by passing hour, minute and second attributes: ', time2)
time1 = time1.replace(hour=17)
print('by replacing the hour attribute in time1: ', time1)

time2 = time2.replace(minute=00)
print('by replacing the minute attribute in time2: ', time2)
```

Output

```
without passing any attribute:  00:00:00
by passing hour, minute and second attributes:  12:55:50
by replacing the hour attribute in time1:  17:00:00
by replacing the minute attribute in time2:  12:00:50
```

Explanation of Code

- In the above code we imported the python datetime library.
- We then saved two different type of time object into the variable (one having no attributes and other with some attributes).
- Then we used replace() to change the value of above generated time.
- Then we print the variable to get different times according to the attributes we passed in the time class.

Datetime

Now, we can use the datetime python class of the datetime module. This class helps to convert the numeric date & time attributes to date-time format in the form of YYYY-MM-DD hr:min:s:ms.

This class accepts 7 attributes Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds in the following order (Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds).

Syntax

```
import datetime

  var1 = datetime.datetime(YYYY, MM, DD, hr, min, s, ms)

  # This will convert the numeral date to the datetime object in the form of
    YYYY-MM-DD hr:min:s:ms.
```

Example

```
import datetime

userdatetime = datetime.datetime(2021, 9, 15, 20, 55, 20, 562789)
```

```
print('userdatetime: ', userdatetime)
print('type of userdatetime: ', type(userdatetime))
```

Output

```
userdatetime:  2021-09-15 20:55:20.562789
type of userdatetime:  <class 'datetime.datetime'>
```

Explanation of Code

- In the above code we imported the datetime library.
- We then saved the date time object into the variable.
- Then we print the variable and also the type of variable whose output is down below.

As we can see that we have changed the numeral python datatime to the datetime object and the same can be verified by the type of object printed below.

Timedelta

Now, we can use the timedelta class of the datetime module.

This class helps to find the difference between the 2 dates/time which is provided by the datetime.timedelta class. No matter whether the difference of dates/time is positive or negative.

This class accepts several arguments like weeks, days, hours, minutes, seconds, milliseconds and microseconds. And, the default value of these arguments is zero.

Syntax

```
import datetime

  var1 = datetime.timedelta(weeks=?, days=?, hours=?, minutes=?, seconds=?,
        milliseconds=?, microsecond=?)

  # This will return the exact date & time object by using all the attributes
    used by the user.
```

Example

```
import datetime

data1= datetime.timedelta(days=1, hours=23, minutes=30)
print('data1: ', data1)

data2= datetime.timedelta(days=4, hours=11, minutes=30)
print('data2: ', data2)
```

```
difference = data2-data1
print('difference between data2 and data1: ', difference)
```

Output

```
data1:  1 day, 23:30:00
data2:  4 days, 11:30:00
difference between data2 and data1:  2 days, 12:00:00
```

Explanation Of Code

- In the above code we imported the datetime library.
- We then saved the date time object into the variable in the form of timedelta format.
- Then we print the print the difference between the 2 days which are generated above.

As we can see that we have printed the difference of the 2 dates below.

Tzinfo

The function .now() that we are using to get the current system time does not have any information about the time zones. In most of the cases there is a need of time zone information. So, in such cases the tzinfo abstract class is used. As it is an abstract class, so it cann't be instantiated directly.

The constructors of datetime object accepts the instances of Tzinfo class.

Naive & Aware Datetime Objects

- Naive Datetime Object: The datetime object which doesn't have any information regarding time zone is considered as naive datetime object.
- The tzinfo value for naive datetime object is NONE.
- Aware Datetime Object: The datetime object which has information regarding time zone is considered as aware datetime object.

Timezone

A time zone represents the standard time which depends upon which part of the world is considered.

datetime.now() returns the local time of the part of world where the user is present. The local time has no relation/information regarding the timezone of that particular region.

We can import one more inbuilt module pytz for using the time zone of different part of the world. Import the module using the below mentioned

code:

```
import pytz
```

Basics of pytz Library

1. We used pytz.utc to get the UTC time into the datetime.datetime.now() function. The +00:00 at the end of the output of UTC time zone represents the UTC offset.
2. We used pytz.timezone('US/CENTRAL') to get the 'US/CENTRAL' time into the datetime.datetime.now() function. The -05:00 at the end of the output of UTC time zone represents the UTC offset of the 'US/CENTRAL' region.

Syntax

```
import datetime

#added another module for using different time zones.
import pytz

#return the local time of that area
var1 = datetime.now()

# return the standard utc time
var2 = datetime.now(pytz.utc)

# return the us/central timezone
var3 = datetime.now(pytz.timezone('US/CENTRAL'))
```

Example

```
import datetime
import pytz


localtime= datetime.datetime.now()
print('local time: ', localtime)

utctime = datetime.datetime.now(pytz.utc)
print('UTC time zone: ', utctime)

ustime = datetime.datetime.now(pytz.timezone('US/CENTRAL'))
print('US time zone: ', ustime)
```

Output

```
local time:  2021-10-15 01:37:39.603581
UTC time zone:  2021-10-14 20:07:39.603716+00:00
US time zone:  2021-10-14 15:07:39.626474-05:00
```

Explanation of Example

- In the above code we have imported the datetime and pytz libraries.
- We then saved different date time object on the basis of time zones into the variable.
- Then we print the variables whose output is down below.

Date Class

.date()

This class helps to convert the numeric date attributes to date format in the form of YYYY-MM-DD.

This class accepts 3 attributes Year, Month & Day in the following order (Year, Month, Day).

Syntax

```
import datetime

  var1 = datetime.date(YYYY, MM, DD)

  # This will convert the numeral date to the date object
```

Example

```
import datetime

dateByUser = datetime.date(2018, 12, 31)
print('dateByUser: ', dateByUser)
print('type of dateByUser: ', type(dateByUser))
```

Output

```
dateByUser:  2018-12-31
type of dateByUser:  <class 'datetime.date'>
```

Explanation of Example

- In the above code we have imported the datetime library.
- We then saved the date time object into the variable.
- Then we print the variable and also the type of variable whose output is down below.

As we can see that we have changed the numeral datatime to the datetime object and the same can be verified by the type of object printed below.

.today()

This function of Date class returns the today's date on the output screen in the format of 'YYYY-MM-DD'.

Example

```
import datetime

print("today's date: ", datetime.date.today())
```

Output

```
today's date: 2021-10-16
```

.min

The .min function returns the earliest representable date. This uses 3 attributes 'MinYear', 'Month' & 'Day' in the following order (datetime.MINYEAR, Month, Day).

Example

```
import datetime

print('earliest year which can be represented: ',
      datetime.date(datetime.MINYEAR, 12, 1))
```

Output

```
earliest year which can be represented:  0001-12-01
```

.max

The .max function returns the latest representable date. This uses 3 attributes 'MaxYear', 'Month' & 'Day' in the following order (datetime.MAXYEAR, Month, Day).

Example

```
import datetime

print('latest year which can be represented: ',
      datetime.date(datetime.MAXYEAR, 2, 25))
```

Output

```
latest year which can be represented:  9999-02-25
```

.day

This function returns the days count of the date which is entered by the user. It gives count from 1 to 30/31 (or precisely number of days in the specified month) both inclusive.

Example

```
import datetime
```

```
dateByUser = datetime.date(2020, 5, 27)
print('day count of the date entered by the user: ', dateByUser.day)
```

Output

```
day count of the date entered by the user:  27
```

.month

This function returns the month count of the date which is entered by the user. It gives count from 1 to 12 both inclusive.

Example

```
import datetime

dateByUser = datetime.date(2024, 2, 29)
print('month count of the date entered by the user:', dateByUser.month)
```

Output

```
month count of the date entered by the user: 2
```

.year

This function returns the year count of the date which is entered by the user. It give count from MINYEAR to MAXYEAR both inclusive.

Example

```
import datetime

dateByUser = datetime.date(2030, 10, 15)
print('year of the date entered by the user:', dateByUser.year)
```

Output

```
year of the date entered by the user: 2030
```

strftime()

This function helps us to represent date in different types of formats such as Short/Long days(Mon, Monday), Short/Long years(21, 2021), Numeric/Alphabetic month(03, March).

The image below represents the notations for the representation of different types of formats of the dates.

Example

```
import datetime

todaydatetime = datetime.datetime.now()
```

```
print('current date and time: ', todaydatetime)
print('fetched out year from current date: ', todaydatetime.strftime('%Y'))
print('fetched out month from current date: ', todaydatetime.strftime('%B'))
print('fetched out day from current date: ', todaydatetime.strftime('%A'))
print('formatted the time from above generated date time: ', todaydatetime.strftime('%H:%M:%S'))
print(todaydatetime.strftime('formatted both date and time: ', "%d/%m/%Y, %H:%M:%S"))
```

Output

```
current date and time:  2021-10-17 12:14:52.394962
fetched out year from current date:  2021
fetched out month from current date:  October
fetched out day from current date:  Sunday
formatted the time from above generated date time:  12:14:52
formatted both date and time: 17/10/2021, 12:14:52
```

Explaination of Example

1. Firstly, we stored the current date and time into our variable 'todaydatetime'.
2. After storing the date we print the actual format of date & time generated.
3. We have used '%Y' for extracting year from the above mentioned date.
4. We have used '%B' for extracting month from the date.
5. We have used '%A' for extracting day from the date.
6. We have use different time formatting formatters '%H, %M & %S' for formatting the time.
7. Lastly, we format both time and date.

Current Date

We have two ways to find out the current date which are mentioned below:

1. .today()
2. .now()

.today()

.today() is used to return the current local date. We just can use it from the date class of datetime module to get the current date.

Code

```
import datetime

currentdate = datetime.date.today()
print('current date: ', currentdate)
```

Output

```
current date:  2021-10-17
```

.now()

.now() is used to return the current local date and time as datetime object. So what we can do is, we can format the datetime object to get only current date.

Code

```
import datetime

currentdate = datetime.datetime.now()

print('raw current date and time generated', currentdate)
print('current date formatted from date and time',
currentdate.strftime("%d/%m/%Y"))
```

Output

```
raw current date and time generated 2021-10-17 12:40:15.974292
current date formatted from date and time 17/10/2021
```

Use of datetime.strptime()

There is a function strptime() of datetime python class of python datetime module which is used for conversion of timestamp string to the datetime python objects.

.strptime() accepts 2 attributes timestamp and format in which we can convert it to datetime object.

Syntax

```
import datetime

var1 = datetime.strptime('TimeStamp', 'Format in which we want to print the datetime object')

TimeStamps are represented as: 2021-01-02T10::52::69.125777
and
Formats are represented as: %Y-%B-%AT%H::%M::%S.%f
```

Example

```
import datetime

datetimeObj = datetime.datetime.strptime('2002-01-02T10::52::59.456777'
,'%Y-%m-%dT%H::%M::%S.%f')

print('changed timestamp string to datetime object', datetimeObj)
print('verifying the type of changed format: ', type(datetimeObj))
```

Output

```
changed timestamp string to datetime object 2002-01-02 10:52:59.456777
```

```
verifying the type of changed format:  <class 'datetime.datetime'>
```

Explanation of Example

- In the above code we have imported the python datetime library.
- We then saved the timestamp and the format of datetime (explained in strftime) object into the variable with the help of strptime().
- Then we print the variable and also the type of variable.

How to Get Current timeStamp?

A timestamp is a sequence of characters used to find when a particular event occurred, generally giving the date and time of the day which is accurate to a small fraction of a second.

Let's see how we can get the current timestamp using the datetime python module.

Syntax

```
import datetime

# Date entered by the user/ Current date
var1 = datetime.now()

var1ToTimestamp = var1.timestamp()
```

**Example*

```
import datetime

timestamp = datetime.datetime.now()

print('current date and time', timestamp)
print('current timestamp', timestamp.timestamp())
```

Output

```
current date and time 2021-10-17 13:43:04.221229
current timestamp 1634458384.221229
```

Explaination Of Code

- In the above code we imported the datetime python library.
- We then saved the date object into the variable.
- Then we used the .timestamp() to convert our date object to the time stamp.

How to Know the Day of the Given Date?

- To get the FullName of the day

Code

```
import datetime

timestamp = datetime.datetime.now()

print('full name of the day of the date used: ', timestamp.strftime('%A'))
```

Output

```
full name of the day of the date used:  Sunday
```

Explaination of Code

- In the above code we imported the datetime python library.
- We then saved the date object into the variable.
- Then we print the full name of the day with the help of strftime() having attribute '%A' as discussed in strftime()
- To get the day count of the date we can use .strftime('%d') onto the date object.

Code

```
import datetime

timestamp = datetime.datetime.now()

print('day count of the day of the date used: ',   timestamp.strftime('%d'))
```

Output

```
day count of the day of the date used:  17
```

Explaination of Code

- In the above code we imported the datetime python library.
- We then saved the date object into the variable.
- Then we print the full name of the day with the help of strftime() having attribute '%d' as discussed in strftime().

Generate a List of Dates from a Given Date Algorithm

Main Function

- Import the datetime python module.
- Get input for the StartDate and the EndDate from the user.
- Run a for loop starting from StartDate to the EndDate(included).
    - while loop is running

        - print the dates which are returned by the UserDefined Function
    - when loop stops
        - get out from it and you get all the between dates on the output screen.

UserDefined Function

- This function takes 2 attributes start date and the end date
- Run a for loop starting from StartDate to the EndDate(included) on the day count.
    - while loop is running
        - return the date to the main function.
    - when loop stops
        - get out form user defined function and returned to the main function.

Code

```
import datetime

def datediff(date1, date2):
    for n in range(int ((date2 - date1).days)+1):
        yield date1 + datetime.timedelta(n)

startdate = datetime.date(2021, 10, 1)
enddate = datetime.date(2021, 10, 11)
print('List of dates between ', startdate.strftime("%d-%m-%y"),
' and ', enddate.strftime("%d-%m-%y"), ' is/are: ')
for date in datediff(startdate, enddate):
    print(date.strftime("%d-%m-%Y"))
```

Output

```
List of dates between  01-10-21  and  11-10-21  is/are:
01-10-2021
02-10-2021
03-10-2021
04-10-2021
05-10-2021
06-10-2021
07-10-2021
08-10-2021
09-10-2021
10-10-2021
11-10-2021
```

Applications Of Python DateTime Module

- Can be used to check the remaining shelf life of items in the warehouse managment softwares.
- Can be used to determine the exact age(in years, months, days) of the person by knowning his/her DateOfBirth.
- Can be used to convert the time to different time zones.

Chapter 5: Python Advanced Topics

1. Python Iterators

Iterators are objects that can be iterated upon. In this tutorial, you will learn how iterator works and how you can build your own iterator using __iter__ and __next__ methods.

Iterators in Python

Iterators are everywhere in Python. They are elegantly implemented within `for` loops, comprehensions, generators etc. but are hidden in plain sight.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python **iterator object** must implement two special methods, `__iter__()` and `__next__()`, collectively called the **iterator protocol**.

An object is called **iterable** if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.

The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

Iterating Through an Iterator

We use the `next()` function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise the `StopIteration` Exception. Following is an example.

```
# define a list
my_list = [4, 7, 0, 3]

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through it using next()

# Output: 4
```

```
print(next(my_iter))

# Output: 7
print(next(my_iter))

# next(obj) is same as obj.__next__()

# Output: 0
print(my_iter.__next__())

# Output: 3
print(my_iter.__next__())

# This will raise error, no items left
next(my_iter)
```

```
4
7
0
3
Traceback (most recent call last):
  File "<string>", line 24, in <module>
    next(my_iter)
StopIteration
```

A more elegant way of automatically iterating is by using the for loop. Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

```
>>> for element in my_list:
...     print(element)
...
4
7
0
3
```

Working of for loop for Iterators

As we see in the above example, the `for` loop was able to iterate automatically through the list.
In fact the `for` loop can iterate over any iterable. Let's take a closer look at how the `for` loop is actually implemented in Python.

```
for element in iterable:
    # do something with element
```

Is actually implemented as.

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

So internally, the `for` loop creates an iterator object, `iter_obj` by calling `iter()` on the iterable.
Ironically, this `for` loop is actually an infinite while loop.
Inside the loop, it calls `next()` to get the next element and executes the body of the `for` loop with this value. After all the items exhaust, `StopIteration` is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

Building Custom Iterators

Building an iterator from scratch is easy in Python. We just have to implement the `__iter__()` and the `__next__()` methods.

The `__iter__()` method returns the iterator object itself. If required, some initialization can be performed.
The `__next__()` method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise `StopIteration`.
Here, we show an example that will give us the next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max=0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration


# create an object
numbers = PowTwo(3)

# create an iterable from the object
i = iter(numbers)

# Using next to get to the next iterator element
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

```
1
2
4
8
Traceback (most recent call last):
```

```
  File "/home/bsoyuj/Desktop/Untitled-1.py", line 32, in <module>
    print(next(i))
  File "<string>", line 18, in __next__
    raise StopIteration
StopIteration
```

We can also use a `for` loop to iterate over our iterator class.

```
>>> for i in PowTwo(5):
...     print(i)
...
1
2
4
8
16
32
```

Python Infinite Iterators

It is not necessary that the item in an iterator object has to be exhausted. There can be infinite iterators (which never ends). We must be careful when handling such iterators.

Here is a simple example to demonstrate infinite iterators.

The built-in function `iter()` can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel
.

```
>>> int()
0

>>> inf = iter(int,1)
>>> next(inf)
0
>>> next(inf)
0
```

We can see that the `int()` function always returns 0. So passing it as `iter(int,1)` will return an iterator that calls `int()` until the returned value equals 1. This never happens and we get an infinite iterator.
We can also build our own infinite iterators. The following iterator will, theoretically, return all the odd numbers.

```
class InfIter:
    """Infinite iterator to return all
       odd numbers"""

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        num = self.num
        self.num += 2
        return num
```

A sample run would be as follows.

```
>>> a = iter(InfIter())
>>> next(a)
1
>>> next(a)
3
>>> next(a)
5
>>> next(a)
7
```

And so on...

Be careful to include a terminating condition, when iterating over these types of infinite iterators.

The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory.

There's an easier way to create iterators in Python. To learn more visit: Python generators using yield.

2. Python Generators

Generators in Python

There is a lot of work in building an iterator in Python. We have to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, and raise `StopIteration` when there are no values to be returned.
This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Create Generators in Python

It is fairly simple to create a generator in Python. It is as easy as defining a normal function, but with a `yield` statement instead of a `return` statement.
If a function contains at least one `yield` statement (it may contain other `yield` or `return` statements), it becomes a generator function.
Both `yield` and `return` will return some value from a function.
The difference is that while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and Normal function

Here is how a generator function differs from a normal function.

- Generator function contains one or more `yield` statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

Here is an example to illustrate all of the points stated above. We have a generator function named `my_gen()` with several `yield` statements.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

An interactive run in the interpreter is given below. Run these in the Python shell to see the output.

```
>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()
```

```
>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1
>>> # Once the function yields, the function is paused and the control is transferred to the caller.

>>> # Local variables and theirs states are remembered between successive calls.
>>> next(a)
This is printed second
2

>>> next(a)
This is printed at last
3

>>> # Finally, when the function terminates, StopIteration is raised automatically on further calls.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that the value of variable `n` is remembered between each call.
Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.

To restart the process we need to create another generator object using something like `a = my_gen()`.
One final thing to note is that we can use generators with for loops directly. This is because a `for` loop takes an iterator and iterates over it using `next()` function. It automatically ends when `StopIteration` is raised.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n
```

```
    n += 1
    print('This is printed at last')
    yield n


# Using for loop
for item in my_gen():
    print(item)
```

When you run the program, the output will be:

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Python Generators with a Loop

The above example is of less use and we studied it just to get an idea of what was happening in the background.

Normally, generator functions are implemented with a loop having a suitable terminating condition.

Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]


# For loop to reverse the string
for char in rev_str("hello"):
    print(char)
```

Output

```
o
l
l
e
h
```

Python Generator Expression

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

Similar to the lambda functions which create anonymous functions, generator expressions create anonymous generator functions.
The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.
The major difference between a list comprehension and a generator expression is that a list comprehension produces the entire list while the generator expression produces one item at a time.

They have lazy execution (producing items only when asked for). For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
list_ = [x**2 for x in my_list]

# same thing can be done using a generator expression
# generator expressions are surrounded by parenthesis ()
generator = (x**2 for x in my_list)

print(list_)
print(generator)
```

Output

```
[1, 9, 36, 100]
<generator object <genexpr> at 0x7f5d4eb4bf50>
```

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object, which produces items only on demand.

Here is how we can start getting items from the generator:

```
# Initialize the list
my_list = [1, 3, 6, 10]

a = (x**2 for x in my_list)
print(next(a))

print(next(a))

print(next(a))

print(next(a))

next(a)
```

When we run the above program, we get the following output:

```
1
9
36
100
Traceback (most recent call last):
  File "<string>", line 15, in <module>
StopIteration
```

Generator expressions can be used as function arguments. When used in such a way, the round parentheses can be dropped.

```
>>> sum(x**2 for x in my_list)
146

>>> max(x**2 for x in my_list)
100
```

Use of Python Generators

There are several reasons that make generators a powerful implementation.

1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2 using an iterator class.

```
class PowTwo:
    def __init__(self, max=0):
        self.n = 0
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

The above program was lengthy and confusing. Now, let's do the same using a generator function.

```
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Since generators keep track of details automatically, the implementation was concise and much cleaner.

2. Memory Efficient

A normal function to return a sequence will create the entire sequence in

memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
```

```
        yield num**2

print(sum(square(fibonacci_numbers(10))))
```

Output

```
4895
```

3. Python Closures

Nonlocal variable in a nested function

Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.

In Python, these non-local variables are read-only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.
Following is an example of a nested function accessing a non-local variable.

```
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    printer()

# We execute the function
# Output: Hello
print_msg("Hello")
```

Output

```
Hello
```

We can see that the nested `printer()` function was able to access the non-local `msg` variable of the enclosing function.

Defining a Closure Function

In the example above, what would happen if the last line of the function `print_msg()` returned the `printer()` function instead of calling it? This means the function was defined as follows:

```
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    return printer  # returns the nested function


# Now let's try calling this function.
# Output: Hello
another = print_msg("Hello")
another()
```

Output

```
Hello
```

That's unusual.

The `print_msg()` function was called with the string `"Hello"` and the returned function was bound to the name `another`. On calling `another()`, the message was still remembered although we had already finished executing the `print_msg()` function.
This technique by which some data (`"Hello` in this case) gets attached to the code is called **closure in Python**.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Try running the following in the Python shell to see the output.

```
>>> del print_msg
>>> another()
Hello
>>> print_msg("Hello")
Traceback (most recent call last):
...
NameError: name 'print_msg' is not defined
```

Here, the returned function still works even when the original function was deleted.

When do we have closures?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.

The criteria that must be met to create closure in Python are summarized in the following points.

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

When to use closures?

So what are closures good for?

Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solution. But when the number of attributes and methods get larger, it's better to implement

a class.

Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier


# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)

# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

Output

```
27
15
30
```

Python Decorators make an extensive use of closures as well.
On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out.

All function objects have a `__closure__` attribute that returns a tuple of cell objects if it is a closure function. Referring to the example above, we know `times3` and `times5` are closure functions.

```
>>> make_multiplier_of.__closure__
>>> times3.__closure__
(<cell at 0x0000000002D155B8: int object at 0x000000001E39B6E0>,)
```

The cell object has the attribute cell_contents which stores the closed value.

```
>>> times3.__closure__[0].cell_contents
3
>>> times5.__closure__[0].cell_contents
5
```

4. Python Decorators

A decorator takes in a function, adds some functionality and returns it. In this tutorial, you will learn how you can create a decorator and why you should use it.

Decorators in Python

Python has an interesting feature called **decorators** to add functionality to an existing code.
This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.

Prerequisites for learning decorators

In order to understand about decorators, we must first know a few basic things in Python.

We must be comfortable with the fact that everything in Python (Yes! Even classes), are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object. Here is an example.

```
def first(msg):
    print(msg)
```

```
first("Hello")

second = first
second("Hello")
```

Output

```
Hello
Hello
```

When you run the code, both functions `first` and `second` give the same output. Here, the names `first` and `second` refer to the same function object.
Now things start getting weirder.

Functions can be passed as arguments to another function.

If you have used functions like `map`, `filter` and `reduce` in Python, then you already know about this.
Such functions that take other functions as arguments are also called **higher order functions**. Here is an example of such a function.

```
def inc(x):
    return x + 1


def dec(x):
    return x - 1


def operate(func, x):
    result = func(x)
    return result
```

We invoke the function as follows.

```
>>> operate(inc,3)
4
>>> operate(dec,3)
2
```

Furthermore, a function can return another function.

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned


new = is_called()

# Outputs "Hello"
new()
```

Output

```
Hello
```

Here, `is_returned()` is a nested function which is defined and returned each time we call `is_called()`.
Finally, we must know about Closures in Python.

Getting back to Decorators

Functions and methods are called **callable** as they can be called.
In fact, any object which implements the special `__call__()` method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.
Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")
```

When you run the following codes in shell,

```
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

In the example shown above, `make_pretty()` is a decorator. In the assignment step:

```
pretty = make_pretty(ordinary)
```

The function `ordinary()` got decorated and the returned function was given the name `pretty`.
We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary).
```

This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the `@` symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like:

```
def divide(a, b):
    return a/b
```

This function has two parameters, `a` and `b`. We know it will give an error if we pass in `b` as 0.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner
```

```
@smart_divide
def divide(a, b):
    print(a/b)
```

This new implementation will return None if the error condition arises.

```
>>> divide(2,5)
I am going to divide 2 and 5
0.4

>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

In this manner, we can decorate functions that take parameters.

A keen observer will notice that parameters of the nested inner() function inside the decorator is the same as the parameters of functions it decorates. Taking this into account, now we can make general decorators that work with any number of parameters.
In Python, this magic is done as function(*args, **kwargs). In this way, args will be the tuple of positional arguments and kwargs will be the dictionary of keyword arguments. An example of such a decorator will be:

```
def works_for_all(func):
    def inner(*args, **kwargs):
        print("I can decorate any function")
        return func(*args, **kwargs)
    return inner
```

Chaining Decorators in Python

Multiple decorators can be chained in Python.

This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner


@star
@percent
def printer(msg):
    print(msg)


printer("Hello")
```

Output

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

The above syntax of,

```
@star
@percent
def printer(msg):
    print(msg)
```

is equivalent to

```
def printer(msg):
    print(msg)
printer = star(percent(printer))
```

The order in which we chain decorators matter. If we had reversed the order as,

```
@percent
@star
def printer(msg):
    print(msg)
```

The output would be:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

5. Python @property decorator

Python programming provides us with a built-in `@property` decorator which makes usage of getter and setters much easier in Object-Oriented Programming.
Before going into details on what `@property` decorator is, let us first build an intuition on why it would be needed in the first place.

Class Without Getters and Setters

Let us assume that we decide to make a class that stores the temperature in

degrees Celsius. It would also implement a method to convert the temperature into degrees Fahrenheit. One way of doing this is as follows:

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32
```

We can make objects out of this class and manipulate the `temperature` attribute as we wish:

```
# Basic method of setting and getting attributes in Python
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32


# Create a new object
human = Celsius()

# Set the temperature
human.temperature = 37

# Get the temperature attribute
print(human.temperature)

# Get the to_fahrenheit method
print(human.to_fahrenheit())
```

Output

```
37
98.60000000000001
```

The extra decimal places when converting into Fahrenheit is due to the floating point arithmetic error. To learn more, visit Python Floating Point Arithmetic Error.

Whenever we assign or retrieve any object attribute like `temperature` as shown above, Python searches it in the object's built-in `__dict__` dictionary attribute.

```
>>> human.__dict__
{'temperature': 37}
```

Therefore, `man.temperature` internally becomes `man.__dict__['temperature']`.

Using Getters and Setters

Suppose we want to extend the usability of the `Celsius` class defined above. We know that the temperature of any object cannot reach below -273.15 degrees Celsius (Absolute Zero in Thermodynamics)
Let's update our code to implement this value constraint.

An obvious solution to the above restriction will be to hide the attribute `temperature` (make it private) and define new getter and setter methods to manipulate it. This can be done as follows:

```
# Making Getters and Setter methods
class Celsius:
    def __init__(self, temperature=0):
        self.set_temperature(temperature)

    def to_fahrenheit(self):
        return (self.get_temperature() * 1.8) + 32

    # getter method
    def get_temperature(self):
        return self._temperature

    # setter method
    def set_temperature(self, value):
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible.")
        self._temperature = value
```

As we can see, the above method introduces two new `get_temperature()` and `set_temperature()` methods. Furthermore, `temperature` was replaced with `_temperature`. An underscore `_` at the beginning is used to denote private variables in Python.

Now, let's use this implementation:

```
# Making Getters and Setter methods
class Celsius:
    def __init__(self, temperature=0):
        self.set_temperature(temperature)

    def to_fahrenheit(self):
        return (self.get_temperature() * 1.8) + 32

    # getter method
    def get_temperature(self):
        return self._temperature

    # setter method
    def set_temperature(self, value):
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible.")
        self._temperature = value


# Create a new object, set_temperature() internally called by __init__
human = Celsius(37)

# Get the temperature attribute via a getter
print(human.get_temperature())

# Get the to_fahrenheit method, get_temperature() called by the method itself
print(human.to_fahrenheit())

# new constraint implementation
human.set_temperature(-300)

# Get the to_fahreheit method
print(human.to_fahrenheit())
```

Output

```
37
98.60000000000001
Traceback (most recent call last):
  File "<string>", line 30, in <module>
  File "<string>", line 16, in set_temperature
ValueError: Temperature below -273.15 is not possible.
```

This update successfully implemented the new restriction. We are no longer allowed to set the temperature below -273.15 degrees Celsius.

Note: The private variables don't actually exist in Python. There are simply norms to be followed. The language itself doesn't apply any restrictions.

```
>>> human._temperature = -300
>>> human.get_temperature()
-300
```

However, the bigger problem with the above update is that all the programs that implemented our previous class have to modify their code from `obj.temperature` to `obj.get_temperature()` and all expressions like `obj.temperature = val` to `obj.set_temperature(val)`.
This refactoring can cause problems while dealing with hundreds of thousands of lines of codes.

All in all, our new update was not backwards compatible. This is where `@property` comes to rescue.

The property Class

A pythonic way to deal with the above problem is to use the `property` class. Here is how we can update our code:

```
# using property class
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32
```

```
    # getter
    def get_temperature(self):
        print("Getting value...")
        return self._temperature

    # setter
    def set_temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible")
        self._temperature = value

    # creating a property object
    temperature = property(get_temperature, set_temperature)
```

We added a `print()` function inside `get_temperature()` and `set_temperature()` to clearly observe that they are being executed.

The last line of the code makes a property object `temperature`. Simply put, property attaches some code (`get_temperature` and `set_temperature`) to the member attribute accesses (`temperature`).

Let's use this update code:

```
# using property class
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    # getter
    def get_temperature(self):
        print("Getting value...")
        return self._temperature

    # setter
    def set_temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible")
        self._temperature = value

    # creating a property object
    temperature = property(get_temperature, set_temperature)
```

```
human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

human.temperature = -300
```

Output

```
Setting value...
Getting value...
37
Getting value...
98.60000000000001
Setting value...
Traceback (most recent call last):
  File "<string>", line 31, in <module>
  File "<string>", line 18, in set_temperature
ValueError: Temperature below -273 is not possible
```

As we can see, any code that retrieves the value of `temperature` will automatically call `get_temperature()` instead of a dictionary (`__dict__`) look-up. Similarly, any code that assigns a value to `temperature` will automatically call `set_temperature()`.
We can even see above that `set_temperature()` was called even when we created an object.

```
>>> human = Celsius(37)
Setting value...
```

Can you guess why?

The reason is that when an object is created, the `__init__()` method gets called. This method has the line `self.temperature = temperature`. This expression automatically calls `set_temperature()`.
Similarly, any access like `c.temperature` automatically calls `get_temperature()`. This is what property does. Here are a few more examples.

```
>>> human.temperature
Getting value
37
>>> human.temperature = 37
Setting value

>>> c.to_fahrenheit()
Getting value
98.60000000000001
```

By using `property`, we can see that no modification is required in the implementation of the value constraint. Thus, our implementation is backward compatible.

The @property Decorator

In Python, `property()` is a built-in function that creates and returns a `property` object. The syntax of this function is:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- `fget` is function to get value of the attribute
- `fset` is function to set value of the attribute
- `fdel` is function to delete the attribute
- `doc` is a string (like a comment)

As seen from the implementation, these function arguments are optional. So, a property object can simply be created as follows.

```
>>> property()
<property object at 0x0000000003239B38>
```

A property object has three methods, `getter()`, `setter()`, and `deleter()` to specify `fget`, `fset` and `fdel` at a later point. This means, the line:

```
temperature = property(get_temperature,set_temperature)
```

can be broken down as:

```
# make empty property
temperature = property()
# assign fget
temperature = temperature.getter(get_temperature)
# assign fset
temperature = temperature.setter(set_temperature)
```

These two pieces of codes are equivalent.

Programmers familiar with Python Decorators can recognize that the above construct can be implemented as decorators.

We can even not define the names `get_temperature` and `set_temperature` as they are unnecessary and pollute the class namespace.
For this, we reuse the `temperature` name while defining our getter and setter functions. Let's look at how to implement this as a decorator:

```
# Using @property decorator
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value
```

```
# create an object
human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

coldest_thing = Celsius(-300)
```

Output

```
Setting value...
Getting value...
37
Getting value...
98.60000000000001
Setting value...
Traceback (most recent call last):
  File "", line 29, in
  File "", line 4, in __init__
  File "", line 18, in temperature
ValueError: Temperature below -273 is not possible
```

The above implementation is simple and efficient. It is the recommended way to use `property`.

6. Python Regular Expressions

Python RegEx

Python has a module named `re` to work with regular expressions. To use it, we need to import the module.

```
import re
```

The module defines several functions and constants to work with RegEx.

re.findall()

The re.findall() method returns a list of strings containing all matches.

Example 1: re.findall()

```
# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'

result = re.findall(pattern, string)
print(result)

# Output: ['12', '89', '34']
```

If the pattern is not found, re.findall() returns an empty list.

re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

Example 2: re.split()

```
import re

string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

If the pattern is not found, re.split() returns a list containing the original string.

You can pass maxsplit argument to the re.split() method. It's the maximum number of splits that will occur.

```
import re

string = 'Twelve:12 Eighty nine:89 Nine:9.'
```

```
pattern = '\d+'

# maxsplit = 1
# split only at the first occurrence
result = re.split(pattern, string, 1)
print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']
```

By the way, the default value of `maxsplit` is 0; meaning all possible splits.

re.sub()

The syntax of `re.sub()` is:

```
re.sub(pattern, replace, string)
```

The method returns a string where matched occurrences are replaced with the content of `replace` variable.

Example 3: re.sub()

```
# Program to remove all whitespaces
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.sub(pattern, replace, string)
print(new_string)

# Output: abc12de23f456
```

If the pattern is not found, `re.sub()` returns the original string.

You can pass `count` as a fourth parameter to the `re.sub()` method. If omited, it results to 0. This will replace all occurrences.

```
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'
replace = ''

new_string = re.sub(r'\s+', replace, string, 1)
print(new_string)

# Output:
# abc12de 23
# f45 6
```

re.subn()

The re.subn() is similar to re.sub() except it returns a tuple of 2 items containing the new string and the number of substitutions made.

Example 4: re.subn()

```
# Program to remove all whitespaces
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.subn(pattern, replace, string)
print(new_string)

# Output: ('abc12de23f456', 4)
```

re.search()

The `re.search()` method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, `re.search()` returns a match object; if not, it returns `None`.

```
match = re.search(pattern, str)
```

Example 5: re.search()

```
import re

string = "Python is fun"

# check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
  print("pattern found inside the string")
else:
  print("pattern not found")

# Output: pattern found inside the string
```

Here, `match` contains a match object.

Match object

You can get methods and attributes of a match object using dir() function.

Some of the commonly used methods and attributes of match objects are:

match.group()

The `group()` method returns the part of the string where there is a match.

Example 6: Match object

```
import re

string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.
match = re.search(pattern, string)

if match:
  print(match.group())
else:
  print("pattern not found")

# Output: 801 35
```

Here, `match` variable contains a match object.
Our pattern `(\d{3}) (\d{2})` has two subgroups `(\d{3})` and `(\d{2})`. You can get the part of the string of these parenthesized subgroups. Here's how:

```
>>> match.group(1)
'801'

>>> match.group(2)
'35'
>>> match.group(1, 2)
('801', '35')

>>> match.groups()
('801', '35')
```

match.start(), match.end() and match.span()

The `start()` function returns the index of the start of the matched substring. Similarly, `end()` returns the end index of the matched substring.

```
>>> match.start()
2
>>> match.end()
8
```

The `span()` function returns a tuple containing start and end index of the

matched part.

```
>>> match.span()
(2, 8)
```

match.re and match.string

The re attribute of a matched object returns a regular expression object. Similarly, string attribute returns the passed string.

```
>>> match.re
re.compile('(\\d{3}) (\\d{2})')

>>> match.string
'39801 356, 2102 1111'
```

We have covered all commonly used methods defined in the re module.

Using r prefix before RegEx

When r or R prefix is used before a regular expression, it means raw string. For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n.
Backlash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

Example 7: Raw string using r prefix

```
import re

string = '\n and \r are escape sequences.'

result = re.findall(r'[\n\r]', string)
print(result)
```

```
# Output: ['\n', '\r']
```

Chapter 6: Error Handling

1. Python Error and In-built Exception in Python

A small typing mistake can lead to an error in any programming language because we must follow the syntax rules while coding in any programming language.

Same is the case with python, in this tutorial we will learn about syntax errors and exceptions in python along with listing down some of the commonly occuring Exceptions in python.

Python: Syntax Error

This is the most common and basic error situation where you break any syntax rule like if you are working with Python 3.x version and you write the following code for printing any statement,

```
print "I love Python!"
```

```
SyntaxError: Missing parentheses in call to 'print'.
```

Because, Python 3 onwards the syntax for using the print statement has changed. Similarly if you forget to add colon(:) at the end of the if condition, you will get a **SyntaxError**:

```
if 7 > 5
   print("Yo Yo!")
```

```
SyntaxError: invalid syntax
```

Hence syntax errors are the most basic type of errors that you will encounter while coding in python and these can easily be fixed by seeing the error message and correcting the code as per python syntax.

Python: What is an Exception?

Contrary to syntax error, exception is a type of error which is caused as a result of malfunctioning of the code during execution.

Your code might not have any syntax error, still it can lead to exceptions when it is executed.

Let's take the most basic example of **dividing a number by zero**:

```
a = 10

b = 0

print("Result of Division: " + str(a/b))
```

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print("Result of Division: " + str(a/b))
ZeroDivisionError: division by zero
```

As we can see in the output, we got **ZeroDivisionError** while the syntax of our python code was absolutely correct, because in this case the error or we should say the exception was generated while code execution.

Python returns a very detailed **exception message** for us to understand the origin point and reason for the exception so that it becomes easier for us to fix our code.

Decoding the Exception Message in Python

The term **Traceback** in the exception message means that python has traced back the code to the point from where the exception occured and will be showing the related messages after this line.

The second line in the exception message, as you can see above, tells us the **name of the python file** and the exact **line number for the code** due to which exception was generated.

If that is still not helpful for someone, in the third line of exception message the complete code statement which lead to the exception is printed.

And then in the last line, python tells us which exception/error occured, which in our example above is **ZeroDivisionError**.

2. Python Exception Handling

Exception handling is a concept used in Python to handle the exceptions and errors that occur during the execution of any program. Exceptions are unexpected errors that can occur during code execution. We have covered about **exceptions and errors in python** in the last tutorial.

Well, yes, exception occur, there can be errors in your code, but why should we invest time in handling exceptions?

The answer to this question is **to improve User Experience**. When an exception occurs, following things happen:

- Program execution abruptly stops.
- The complete exception message along with the file name and line number of code is printed on console.
- All the calculations and operations performed till that point in code are lost.

Now think that one day you are using **Studytonight**'s website, you click on some link to open a tutorial, which, for some unknown reason, leads to some exception. If we haven't handled the exceptions then you will see exception message while the webpage is also not loaded. Will you like that?

Hence, exception handling is very important to handle errors gracefully and displaying appropriate message to inform the user about the malfunctioning.

Handling Exceptions using try and except

For handling exceptions in Python we use two types of blocks, namely, try block and except block.

In the try block, we write the code in which there are chances of any error or exception to occur.

Whereas the except block is responsible for catching the exception and executing the statements specified inside it.

Below we have the code performing **division by zero**:

```
a = 10

b = 0

print("Result of Division: " + str(a/b))
```

```
Traceback (most recent call last):
File "main.py", line 3, in <module>
print("Result of Division: " + str(a/b))
ZeroDivisionError: division by zero
```

The above code leads to exception and the exception message is printed as output on the console.

If we use the try and except block, we can handle this exception gracefully.

```
# try block
try:
    a = 10
    b = 0
    print("Result of Division: " + str(a/b))
except:
    print("You have divided a number by zero, which is not allowed.")
```

The try block

As you can see in the code example above, the try block is used to put the whole code that is to be executed in the program(which you think can lead to exception), if any exception occurs during execution of the code inside the try block, then it causes the execution of the code to be directed to the except block and the execution that was going on in the try block is interrupted. But, if no exception occurs, then the whole try block is executed and the except block is never executed.

The except block

The try block is generally followed by the except block which holds the exception cleanup code(*exception has occured, how to effectively handle the situation*) like some print statement to **print some message** or may be **trigger some event** or **store something in the database** etc.

In the **except block**, along with the keyword except we can also provide the **name of exception class** which is expected to occur. In case we do not provide any exception class name, it catches all the exceptions, otherwise it will only catch the exception of the type which is mentioned.

Here is the **syntax**:

```
# except block

except(<Types of Exceptions to catched>):

    # except block starts
```

If you notice closely, we have mentioned **types of exceptions**, yes, we can even provide names of multiples exception classes separated by comma in the **except** statement.

Code Execution continues after except block

Another important point to note here is that code execution is interrupted in the try block when an exception occurs, and the code statements inside the try block after the line which caused the exception are not executed.

The execution then jumps into the except block. And after the execution of the code statements inside the except block the code statements after it are executed, just like any other normal execution.

Let's take an example:

```
# try block

try:

    a = 10

    b = 0

    print("Result of Division: " + str(a/b))

    print("No! This line will not be executed.")

except:

    print("You have divided a number by zero, which is not allowed.")



# outside the try-except blocks

print("Yo! This line will be executed.")
```

```
You have divided a number by zero, which is not allowed.

Yo! This line will be executed.
```

Catching Multiple Exceptions in Python

There are multiple ways to accomplish this. Either we can have multiple except blocks with each one handling a specific exception class or we can handle multiple exception classes in a single except block.

Multiple except blocks

If you think your code may generate different exceptions in different situations and you want to handle those exceptions individually, then you can have multiple except blocks.

Mostly exceptions occur when user inputs are involved. So let's take a simple example where we will ask user for two numbers to perform division operation on them and show them the result.

We will try to handle multiple possible exception cases using multiple except blocks.

Try running the above code, provide **0** as value for the denominator and see what happens and then provide some **string**(non-integer) value for any variable. We have handled both the cases in the above code.

Handling Multiple Exceptions with on except block

As you can see in the above example we printed different messages based on what exception occured. If you do not have such requirements where you need to handle different exception individually, you can catch a set of exceptions in a single exception block as well.

Here is above the above code with a single except block:

```
# try block

try:

    a = int(input("Enter numerator number: "))

    b = int(input("Enter denominator number: "))

    print("Result of Division: " + str(a/b))

# except block handling division by zero

except(ValueError, ZeroDivisionError):
```

```
    print("Please check the input value: It should be an integer greater than 0")
```

here we have handled both the exceptions using a single except block while showing a meaningful message to the user.

Generic except block to Handle unknown Exceptions

Although we do try to make our code error free by testing it and using exception handling but there can be some error situation which we might have missed.

So when we use except blocks to handle specific exception classes we should always have a **generic** except block at the end to handle any runtime excpetions(surprise events).

```
# try block
try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number: "))
    print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ZeroDivisionError):
    print("You have divided a number by zero, which is not allowed.")
# except block handling wrong value type
except(ValueError):
    print("You must enter integer value")
# generic except block
except:
    print("Oops! Something went wrong!")
```

In the code above the first except block will handle the **ZeroDivisionError**,

second except block will handle the **ValueError** and for any other exception that might occur we have the third except block.

In the coming tutorials we will learn about finally block and how to raise an exception using the raise keyword.

3. Exeption Handling: Finally

The finally code block is also a part of exception handling. When we **handle exception using the try and except block**, we can include a finally block at the end. The finally block is always executed, so it is generally used for doing the concluding tasks like closing file resources or closing database connection or may be ending the program execution with a delightful message.

finally block with/without Exception Handling

If in your code, the except block is unable to catch the exception and the exception message gets printed on the console, which interrupts code execution, still the finally block will get executed.

Let's take an example:

Try to run the above code for two different values:

Enter some integer value as numerator and provide **0** value for denominator. Following will be the output:

```
1. You have divided a number by zero, which is not allowed.
2. Code execution Wrap up!
3. Will this get printed?
```

As we have handled the **ZeroDivisionError** exception class hence first the except block gets executed, then the finally block and then the rest

of the code.

Now, some integer value as numerator and provide some **string** value for denominator. Following will be the output:

```
Code execution Wrap up!
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    b = int(input("Enter denominator number: "))
ValueError: invalid literal for int() with base 10: 'dsw'
```

As we have not handled the **ValueError** exception, hence our code will stop execution, exception will be thrown, but still the code in the finally block gets executed.

Exception in except block

We use the except block along with try block to handle exceptions, but what if an exception occurred inside the except block. Well the finally block will still get executed.

```
# try block
try:
    a = 10
    b = 0
    print("Result of Division: " + str(a/b))
# except block handling division by zero
except(ZeroDivisionError):
    print("Result of Division: " + str(a/b))
finally:
```

```
  print("Code execution Wrap up!")
```

```
Code execution Wrap up!
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    print("Result of Division: " + str(a/b))
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print("Result of Division: " + str(a/b))
ZeroDivisionError: division by zero
```

Clearly the finally block was the first to be printed on the console follwed by the first exception message and then the second exception message.

4. Python Exception Handling: raise Keyword

While the try and except block are for handling exceptions, the raise keyword on the contrary is to **raise an exception**.

Following is the **syntax**:

```
raise EXCEPTION_CLASS_NAME
```

Taking a simple usage example:

```
raise ZeroDivisionError
```

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    raise ZeroDivisionError
ZeroDivisionError: division by zero
```

If you have a piece of code where along with exception handling you have put in place some conditional statements to validate input etc, then in case of the conditions failing we can either just print a message or simple raise an exception which can then get handled by the common exception handling mechanism.

See the code below,

```
a = 10
b = 9
try:
    print(a/b)
except ZeroDivisionError:
    print("Please enter valid integer value")
```

Consider the above code where we have handled ZeroDivisionError, in this code we want to add a new validation for restricting user from inputting negative values.

Then we can simply add a new condition and use the raise keyword to raise an exception which is already handled.

```
a = 10

b = 9

try:

  # condition for checking for negative values

  if a < 0 or b < 0:

    # raising exception using raise keyword

    raise ZeroDivisionError

  print(a/b)

except ZeroDivisionError:

  print("Please enter valid integer value")
```

By this example we want to explain to you why and where we should use raise keyword to explicitly raise an exception.

raise Without Specifying Exception Class

When we use the raise keyword, it's not necessary to provide an exception class along with it. When we don't provide any exception class name with the raise keyword, it reraises the exception that last occured.

This is used generally inside an except code block to reraise an exception which is catched.

For example

```
a = 10

b = 0

try:

  print(a/b)

except ZeroDivisionError:
```

```
  raise
```

```
Traceback (most recent call last):
  File "main.py", line 4, in
    print(a/b)
ZeroDivisionError: division by zero
```

raise With an Argument

We can also provide an argument while raising an exception which is displayed along with the exception class name on the console.

We can pass any string for describing the reason for the exception or anything else.

```
raise ValueError("Incorrect Value")
```

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    raise ValueError("Incorrect Value")
ValueError: Incorrect Value
```

Chapter 7: Multithreading

1. Introduction to Multithreading In Python

Threads

Threads are lightweight processes (subparts of a large process) that can run concurrently in parallel to each other, where each thread can perform some task. Threads are usually contained in processes. More than one thread can exist within the same process. Within the same process, threads share memory and the state of the process.

Types Of Thread

There are two kinds of thread:

- Kernel-level threads
- User level threads

Below we have explained a few differences between the two:

What is Multithreading?

Now that we have a basic idea about what threads are, let's try to understand the concept of Multithreading.

Modern computers have a CPU that has multiple processing cores and each of these cores can run many threads simultaneously which gives us the ability to perform several tasks concurrently. This process of running multiple Threads concurrently to perform tasks parallely is called **Multithreading**.

Multithreading provides the following benefits:

- Multiple threads within a process share the same data space and can, therefore, share information or communicate with each other more easily than if they were separate processes.
- Threads do not require much memory overhead; they are cheaper than processes in terms of memory requirements.
- Multithreaded programs can run faster on computer systems with multiple CPUs because these threads can be executed concurrently.

Time for an Example

Let's say you create a simple application for an Event's registration where attendees must register if they wish to attend the event. You have a simple HTML form for the users to fill in, and a backend which is a single threaded application.

As the application is single threaded, it can only process one request at a time. But what if the event is a "Coldplay's Music Concert" where millions of people want to register. Processing one request at a time will drastically slow down the performance.

So, we make the application multi-threaded and start multiple threads inside it hence allowing parallel processing.

Multithreading in Python

For performing multithreading in Python threading module is used.The threadingmodule provides several functions/methods to implement multithreading easily in python.

Before we start using the threading module, we would like to first introduce you to a module named time, which provides a time(), ctime() etc functions which we will be using very often to get the current system time and another crucial function sleep() which is used to suspend the execution of the current thread for a given number of seconds.

For example,

Now let's see how we can use the threading module to start multiple threads.

Code Example:

Let's try to understand the above code:

We imported the thread class using import threading statement and we also imported the time module. To create a new thread, we create an object of te Thread class. It takes the following arguments:

target: The function which will be executed by the thread.

args: The arguments to be passed to the target function. We can pass multiple arguments separated by comma.

In the example above, we created 2 threads with different target functions i.e. thread1(i) and thread2(i).

To start a thread, we have used the start() method of the Thread class.

We have also used the time.sleep() method of the time module to pause the execution of thread1 for 3 seconds.

Once the threads start, the current program (you can think of it as the main thread) also keeps on executing. In order to prevent the main program from completing its execution until the thread execution is complete, we use the join() method.

As a result, the current program will wait for the completion of t1and t2 and only after their execution is finished, the remaining statements of the current program will get executed i.e the statement print('Execution completed.').

You should try running the above code once after commenting out the code on line number 16,17 where we use the join method and see the result.

So with this our basic introduction to multithreading in python is completed and now we know how to create and start multiple threads in python.

2. Threading Module In Python

As we have seen in the previous tutorial, threading module is used for creating, controlling and managing threads in python. In this tutorial, we will discuss about various functions and object types defined by the threading module.

threading Module Functions

This module provides the following functions for managing threads:

Here is the code snippet from last tutorial, we will be using this to see the various functions provided by the threading module in action.

threading.active_count() Function

This function returns the number of Thread objects currently alive.

```
import time
import threading

def thread1(i):
    time.sleep(3)
    #print('No. printed by Thread 1: %d' %i)
def thread2(i):
    time.sleep(3)
    #print('No. printed by Thread 2: %d' %i)

if __name__ == '__main__':
    t1 = threading.Thread(target=thread1, args=(10,))
    t2 = threading.Thread(target=thread2, args=(12,))
    t1.start()
    t2.start()
    print("No. of active threads: " + threading.active_count())
    t1.join()
    t2.join()
```

```
No. of active threads: 3
```

Try running the this code in the terminal above. You will see the number of thread count to be **3**, because we have created 2 threads and there in the main thread in which the complete execution is taking place.

threading.current_thread()

This function will return the current **Thread** object, corresponding to the caller's thread of control(which is in the control of caller currently). If the caller's thread of control was not created through the **threading** module(for example the **main** thread), then a dummy thread object with limited functionality is returned.

```
import time
import threading

def thread1(i):
    time.sleep(3)
    #print('No. printed by Thread 1: %d' %i)

def thread2(i):
    time.sleep(3)
    #print('No. printed by Thread 2: %d' %i)

if __name__ == '__main__':
    t1 = threading.Thread(target=thread1, args=(10,))
```

```
    t2 = threading.Thread(target=thread2, args=(12,))
    t1.start()
    t2.start()
    print("Current thread is: " + threading.current_thread())
    t1.join()
    t2.join()
```

```
Current thread is: <_MainThread(MainThread, started 16152)>
```

threading.get_ident()

This function returns the **thread identifier** of the current thread. This is a nonzero integer value. If we started the thread, then this method will return its identifier, otherwise, it will return **None**.

We can use this method to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits(stops) and another thread is created.

```
threading.get_ident()
```

```
140578859194112
```

threading.enumerate()

This method returns a list of all Thread objects currently alive. The list includes **daemonic** threads(when the program quits, all the daemon threads associated with it are killed automatically), **dummy** thread objects created by

the current thread, and the **main** thread.

Terminated threads and threads that have not yet been started are not present in this list.

```
threading.enumerate()
```

```
[<_MainThread(MainThread, started 139890175817472)>, <Thread(Thread-1, started 139890151225088)>, <Thread(Thread-2, started 139890142832384)>]
```

threading.main_thread()

This method returns the **main** Thread object. In normal conditions, the **main** thread is the thread from which the python interpreter was started.

```
threading.main_thread()
```

```
<_MainThread(MainThread, started 139890175817472)>
```

threading.settrace(fun)

This method is used to set a trace function/hook for all the threads started using the threadingmodule. The trace funtion is passed to sys.settrace() method for each thread, which is attache to the thread before the run() method is called.

The callback function which we want to act as the trace function will receive three arguments, frame (the stack frame from the code being run), event (a string naming the type of notification), and arg (an event-specific value)

```
def hello(frame, event, arg):
    print("Hello, I am a trace hook.")
threading.settrace(hello)
```

Try updating the code in the terminal at the top. Put the hello function outside the main method and the statement threading.settrace(hello) just above the

statement where we call the start method for the threads.

threading.setprofile(fun)

This method is used to set a profile function for all threads started from the threading module. Just like the trace funtion, profile function is also passed to sys.setprofile() method for each thread, which is attache to the thread before the run() method is called.

```
threading.setprofile(hello)
```

threading.stack_size([size])

This method returns the thread stack size utilised when creating new threads. The size argument is optional and can be used to set the stack size to be used for creating subsequent threads, and must be **0** or a positive integer (D=default value is 0).

If changing the thread stack size is unsupported, a **RuntimeError** is raised.

If the specified stack size is invalid, a **ValueError** is raised.

Currently **32 KiB** is the minimum stack size which is supported to provide enough stack space for the interpreter.

threading.TIMEOUT_MAX

Apart from the above specified functions, the threading module also defines a constant. If you specify a timeout value which is greater than the value of the TIMEOUT_MAX constant, you will get an OverflowError.

threading Module Objects

Apart from the functions specified above, the threading module also provides

many classes whose objects are very useful in creating and managing threads.

The above table gives a brief introduction of various object types in python threading module. We will discuss about all these objects in details in the next few tutorials.

3. Thread class and its Object - Python Multithreading

In the threading module the most popular and the most used call is the Thread class, which is primarily used to create and run threads. Thread class provides all the major functionalities required to create and manage a thread.

Thread objects are the objects of the Thread class where each object represents an activity to be performed in a separate thread of control.

There are two ways to create the Thread object and specify the activity to be performed:

- by passing a callable object to the constructor
- or, by overriding the run() method in a subclass.

Thread object which is created using constructor or run method can be started by using start() method. Whenever a Thread object starts a new thread then internally it's run() method is invoked.

Here is a simple example:

In the example above, we have also used the time module to make one of the thread sleep.

Basic syntax of the Thread class constructor is:

```
Thread(group=None, target=None, name=None, args=(), kwargs={})
```

We will explain the arguments of the Thread class constructor in the section below.

How Thread works?

Once we initialize a thread using the Thread class constructor, we must call its start() method to start the thread.

When the thread starts, the thread is considered **alive** and **active**. When its run() method terminates, either normally, or due to an unhandled exception then the thread stops being alive or active. The isAlive() method tests whether the thread is alive or not at any given point of time.

Other threads can call a thread's join() method to join any thread. This blocks the calling thread until the thread whose join() method is called is terminated.

For example, in the code above, from the **main** thread, we call t1.join() and t2.join(), hence the **main** thread will wait for the threads t1 and t2 to terminate and then end.

Every thread has a name associated with it. The name can be passed to the constructor, or we can set or retrieve name by using setname() and getname() methods respectively.

A flag **daemon thread** can be associated with any thread. The significance of this flag is that the entire python program exits when only daemon threads are left. The flag can be set or retrieved by using setDaemon() method and getDaemon() method respectively.

The **main thread** object corresponds to the initial thread of control in the python program. It is not a daemon thread.

Functions and Constructor in the Thread class

Now that we have seen a basic threading program with threads running, it's time to understand the code along with exploring all the important methods provided by the Thread class.

Thread class Constructor

Following is the basic syntax of the Thread class constructor:

```
Thread(group=None, target=None, name=None, args=(), kwargs={})
```

The constructor allows many arguments, some of which are required while some are not. Let's see what they are:

- group: Should be **None**. It is reserved for future extension.
- target: This is the callable object or task to be invoked by the run() method. As you can see in the code example at the top, we have specified the function names **thread1** and **thread2** as the value for this argument. It's default value is **None**.
- name: This is used to specify the thread name. By default, a unique name is generated following the format **Thread-N**, where **N** is a small decimal number.
- args: This is the argument **tuple** for the target invocation. We can provide values in it which can be used in the traget method. It's default value is empty, i.e. ()
- kwargs: This is keyword argument **dictionary** for the target invocation. This defaults to {}.

start() method

This method is used to start the thread's activity. When we call this method, internally the run() method is invoked which executes the target function or the callable object.

run() method

Method representing the thread's activity.

You may override this method in a subclass extending the Thread class of the threading module. The standard run() method invokes the callable object passed to the object's constructor as the target argument with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

Here we have a simple example, where we have created a subclass in which we will override the run() method.

join([timeout]) method

When we call this method for any thread, it blocks the calling thread until the thread whose join() method is called terminates, either normally or through an unhandled exception.

If you want to provide the timeout argument, it should be a floating point number.

getName() method

This method is used to return the thread's name.

setName(name) method

Used for setting the thread's name. The name is a string used for identification purposes only.

isAlive() method

This method returns whether the thread is alive or not. A thread is alive from the moment the start() method returns until its run() method terminates.

isDaemon() method

This method is used to get the value of the thread's daemon flag.

setDaemon(daemonic) method

This method is used to set the thread's daemon flag to the Boolean value **daemonic**. This must be called before start() method is called.

The entire Python program exits when no active non-daemon threads are left.

4. Thread Synchronization using Event Object

It's time to learn more about threads in python. In this tutorial we will be covering an important classe, the Event class which is used for thread synchronization in python.

This class is used for inter thread communication by generating events.

Python Multithreading: Event Object

The Event class object provides a simple mechanism which is used for communication between threads where one thread signals an event while the other threads wait for it. So, when one thread which is intended to produce the signal produces it, then the waiting thread gets activated.

An internal flag is used by the event object known as the **event flag** which can be set as true using the set() method and it can be reset to false using the clear() method.

The wait() method blocks a thread until the event flag for which it is waiting is set true by any other thread..

Following are the useful functions used along with an event object:

Initialize Event object

We can initialize an Event object as follows:

```
import threading

are_you_coming = threading.Event()
```

When we initialize an event object like this the internal flag is set to **false** by default.

isSet() method

This method returns true if and only if the internal flag is true.

```
import threading

are_you_coming = threading.Event()

print(are_you_coming.isSet())
```

set() method

When this method is called for any event object then the internal flag is set to true. And as soon as set() methos is called for any event all threads waiting for it are awakened.

clear() method

This method resets the internal flag to false. Subsequently, threads calling wait() on the event for which clear() is called, it will block until the internal flag is not true again.

wait([timeout]) method

When we have to make any thread wait for an event, we can do so by calling this method on that event which has the internal flag set to false, doing so blocks the thread until the internal flag is true for the event.

If the internal flag is true on entry, then the thread will never get blocked. Otherwise, it is blocked until another thread calls set() to set the flag to true, or until the optional timeout occurs. The timeout argument specifies a timeout for the operation in seconds.

Time for an Example

Let's have a simple code example to demonstrate the usage of Event class object.

In the code below we will create a thread and make it wait for an event which will be generated by the **main** thread, releasing the first thread.

In the above program we have also used the timeout property of the wait() method.

When a thread calls wait([timeout]) method, the method returns a boolean value **true** if the wait is released on receiving the event object, else it returns **false** if the wait is released due to timeout.

To test this, change the value of timeout sent as argument args=(e,4) on line 18 and keep it less than the sleep time, for example set the timeout value to 2

and see the output.

5. Timer Object - Python Multithreading

Timer objects are created using Timer class which is a subclass of the Thread class. Using this class we can set a delay on any action that should be run only after a certain amount of time has passed(timer) and can easily be cancelled during that delay.

Timers are started, just like normal threads, by calling their start() method. A timer thread can be stopped (before its action has begun) by calling its cancel() method.

Timer object is generally used to implement scheduled tasks which supposed to be executed only after a certain instant of time.

Also, its not necessary that the Timer object will be executed exactly after the scheduled time as after that the python intrepreter looks for a thread to execute the timer object task, which if not available leads to more waiting.

Syntax for creating Timer object

Following is the syntax for the Timer class constructor:

```
threading.Timer(interval, function, args=[], kwargs={})
```

This way we can create a timer object that will run the **function** with arguments args and keyword arguments kwargs, after interval seconds have passed.

Methods of Timer class

In the Timer class we have two methods used for starting and cancelling the execution of the timer object.

start() method

This method is used to start the execution of the timer object. When we call this method, then the timer object starts its timer.

cancel() method

This method is used to stop the timer and cancel the execution of the timer object's action. This will only work if the timer has not performed its action yet.

Time for an Example

Below we have a simple example where we create a timer object and start it.

```
import threading

def task():
    print("timer object task running...")

if __name__=='__main__':
    t = threading.Timer(10, task)
    t.start() # after 10 seconds, task will be executed
```

The above program is a simple program, now let's use the cancel method to cancel the timer object task's execution.

In the program above, first comment the code on line number 13 and 14 and

run the program, then uncomment those lines and see the cancel() method in action.

6. Condition Object - Thread Synchronization in Python

In order to synchronize the access to any resources more efficiently, we can associate a condition with tasks, for any thread to wait until a certain condition is met or notify other threads about the condition being fulfilled so that they may unblock themselves.

Let's take a simple example to understand this. In the **Producer Consumer problem**, if there is one Produces producing some item and one Consumer consuming it, then until the Producer has produced the item the Consumer cannot consume it. Hence the Consumer waits until the Produced produces an item. And it's the duty of the Producer to inform the Consumer that an item is available for consumption once it is successfully produced.

And if there are multiple Consumers consuming the item produced by the Producer then the Producer must inform all the Consumers about the new item produced.

This is a perfect usecase for the condition object in multithreading in python.

Condition object: wait(), notify() and notifyAll()

Now that we know what the condition object is used for in python multithreading, let's see the syntax for it:

```
condition = threading.Condition([lock])
```

The condition object takes in an optional lock object as argument. If we do not provide anything then it creates a default lock.

A condition object has acquire() and release() methods that call the

corresponding methods of the associated lock. It also has a wait() method, and notify() and notifyAll() methods. These three must only be called after the calling thread has acquired the lock.

Condition class methods

Following are condition class methods:

acquire(*args) method

This method is used to acquire the lock. This method calls the corresponding acquire() method on the underlying lock present in the condition object; The return value is whatever that method returns.

release() method

this method is used to release the lock. This method calls the corresponding release() method on the underlying lock present in the condition object.

wait([timeout]) method

This method is used to block the thread and make it wait until some other thread notifies it by calling the notify() or notifyAll() method on the same condition object or until the timeout occurs.

This must only be called when the calling thread has acquired the lock.

When called, this method releases the lock and then blocks the thread until it is awakened by a notify() or notifyAll() call for the same condition variable from some other thread, or until the timeout occurs.

This method returns True if it is released because of notify() or notifyAll() method else if timeout occurs this method will return False boolean value.

notify() method

It wakes up any thread waiting on the corresponding condition. This must only be called when the calling thread has acquired the lock. Also, calling this method will wake only one waiting thread.

notifyAll() method

It wakes up all the threads waiting on this condition. This method acts like notify() method, but wakes up all the waiting threads instead of one.

Time for an Example!

In the code example below we have implemented a simple producer-consumer solution, where the producer produces an item and adds it to a list from which the consumer is consuming the items.

Few important takeaways from the code example above:

1. We have created a class SomeItem which has a list which acts as the shared resource between the producer and consumer thread.
2. The **producer thread** is randomly generating some list items and adding it to the list.
3. The **consumer thread** tries to consume the item, if item is not found, it starts to wait. If the producer sends a notification to the consumer about the item creation before its timeout, then the consumer consumes the item else it exits due to timeout.

This is a very simple example covering all the usecases of condition object. Try running the above program with 2 consumer threads and single producer thread.

7. Barrier Object - Python Multithreading

Barrier object is created by using Barrier class which is available in the threading module. This object can be used where we want a set of threads to wait for each other.

For example, if we have two threads and we want both the threads to execute when both are ready. In such situation both the threads will call the wait() method on the barrier object once they are ready and both the threads will be released simultaneously only when both of them have called the wait() method.

Following is the syntax of the Barrier class to initialize a barrier object:

```
threading.Barrier(parties, action=None, timeout=None)
```

Where, parties specifies the number of threads waiting at the barrier, action can be used to specify a function which will be executed by any one thread waiting for the barrier and timeout is used to specify a timeout value in seconds after which the barrier will be released from all the waiting threads.

Functions provided by Barrier class

Following are the function provided by the Barrier class:

wait(timeout=None) method

When we want a set of threads to wait for each other, we initialise the barrier object with the number of threads specified as parties parameter while barrier object creation.

Then, the barrier will be released only when the same number of threads call the wait() method of the barrier object.

If in a thread we provide the timeout value while calling the wait() method, then that thread will get released from the barrier once the timeout time is passed.

This method returns an integer value from **0** to **parties-1**. We can use this

value to identify which all threads have reached the waiting point of the barrier and which all are still not there, for example:

```
i = barrier.wait()

if i == 0:

    # Only one thread Will be allowed to print this

    print("0th thread passed the barrier")
```

The return value of the wait() method can be used for carrying out some clean up task for example if we have 3 threads doing some work and using a temporary file to store data, once the threads are done, we can put a check on the wait() method that when the last thread reaches its waiting point and the barrier is to be released, before that delete the file.

```
i = barrier.wait()

if i == 2:

    # delete the temporay file
```

If the wait call times out, the barrier is put into the **broken state**.

The wait() method may raise a BrokenBarrierError if the barrier breaks or resets as a thread waits.

reset() method

This function resets the barrier to its default, empty state. If there are threads waiting for the barrier to get released will receive BrokenBarrierError.

abort() method

This method when called on a barrier puts it into the **broken state**. Once this method is called by any of the waiting thread, the rest of the threads waiting for the barrier to be released will receive BrokenBarrierError.

We may want to use this method in case of some deadlock situation to release the waiting threads.

parties

This returns the number of threads we need to pass the barrier.

n_waiting

This returns the number of threads that currently are waiting for the barrier to be released.

broken

This is a Boolean value that is True if the barrier is in the **broken state**.

Time for an Example!

Below we have a simple example where we have two threads representing **server** and **client** where in the client thread will wait for the server to get ready before sending any request to it.

In the above code, try calling the abort() method in either client or server and see what happens. You might have to use the try block and catch the BrokenBarrierError in the code.

Chapter 8: Python Logging

1. Python Logging Basic Configurations

To **configure** the python logging module, to set the log level, log format, etc., we can use the basicConfig(**kwargs) method where **kwargs in the function definition means this function takes **variable length arguments**, which should be passed in the key-value form.

Some commonly used parameters in the basicConfig() function is given below:

- level: With the help of **this parameter the root logger** will be **set to the specified severity level, like DEBUG, INFO, ERROR, etc.**
- filename: This parameter is mainly used to specify the **file name** if we want to store the logs in a file.
- filemode: If the filename is given, the **file is opened in this mode**. The default mode is a, which means **append**.
- format: This is used to **specify the format of the log message**. For example, if you want to add timestamp with the logs, or the name of the python script or maybe the function or class name in the logs, then we can specify the appropriate format for that.

In the basicConfig() function we can either provide the **level** parameter with an appropriate value, **filename** and **filecode** for printing logs in file, and the **format** parameter to specify the logs, or you can specify all the parameters together, that is the beauty of **kwargs, which means the number of arguments that can be supplied to a function is not fixed.

Let's take an Example

Let us take a look at the example for the clear understanding of this method:

```
import logging
```

```
logging.basicConfig(level=logging.INFO)

logging.info('This is an info message.This will get logged.')
```

```
INFO:root: This is an info message. This will get logged.
```

All events at or above INFO level will now get logged.

Points to remember:

- It is important to note here that calling basicConfig() to **configure the root logger** works in the case only if the root logger has not been configured before. Basically, this function can only be called once.
- Also debug(), info(), warning(), error(), and critical(), all these functions internally call basicConfig() method without any arguments automatically **if it has not been called before**.
- This means that after the first time one of the above-given functions is called, you can no longer **configure the root logger** because they would have called the basicConfig() function internally.

Thus the default setting in basicConfig() is to set the logger to write to the console in the following format:

```
ERROR:root: This is an error message
```

With log level, logger name and then the log message.

Store Logs in File

We can use the basicConfig() method to store logs in a file. How we can do so, we will learn in the next tutorial where we will cover how to store Python code logs in a file.

Set Format of Logs

We can configure the logging module to print logs in a any format we want. There are some standard formats which are universally used, and we can

configure the python logger to print logs in those formats.

There are some **basic components** of logs that are already a part of the **LogRecord** and we can easily add them or remove them from the output format.

Let's take a few examples to understand this.

Add process ID to logs with loglevel and message

If you want to **include the process ID** for the running python script along with the **log level and message** then the code snippet for the same is given below:

```
import logging

logging.basicConfig(format='%(process)d-%(levelname)s-%(message)s')
logging.warning('This message is a warning')
```

```
1396-WARNING-This message is a warning
```

In the code above, in the basicConfig() method, we have set the format parameter with the format string as value in which we have specified, what components we want in our log, along with specifying its **datatype**, like d with **process** to print the integer process Id, then s for **loglevel** which is string value, and same for the **message**

Also, in the code above, we can set the format with the LogRecord attributes set in any order.

Add Timestamp to logs with log message

You can also add **date and time info(timestamp)** to your logs along with the log message. The code snippet for the same is given below:

```
import logging
```

```
logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)

logging.info('This message is to indicate that Admin has just logged in')
```

```
2020-06-19 11:43:59,887 - This message is to indicate that Admin has just logged in
```

In the above code %(asctime)s adds the time of the creation of the LogRecord. Also, we have configured the log level too with code level=logging.INFO.

Use the datefmt attribute

You can also **change the format** using the datefmt attribute, which uses the **same formatting language** as the formatting functions in the **datetime module**, such as time.strftime():

```
import logging


logging.basicConfig(format='%(asctime)s - %(message)s', datefmt='%d-%b-%y %H:%M:%S')

logging.warning('The Admin just logged out')
```

```
19-Jun-20 11:50:28 - The Admin just logged out
```

As you can see the log printed as the output of the above code has a date format **DD-MMM-YY** along with time.

So, using the basicConfig(**kwargs) method, we can configure the logs format as we want, like adding timestamp to logs, adding process id to logs, printing log level along with logs and log messages.

2. Python - Print Logs in a File

If you want to print python logs in a **file** rather than on the **console** then we can do so using the basicConfig() method by providing filename and filemode as parameter.

The **format of the message** can be specified by using format parameter in basicConfig() method.

Let us take a basic example to print logs to a file rather than on the console. The code snippet is given below:

```
import logging    # first of all import the module


logging.basicConfig(filename='std.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')

logging.warning('This message will get logged on to a file')
```

```
root - ERROR - This message will get logged on to a file
```

The above output shows **how the message will look like** but keep in mind it will be written to a **file named std.log** instead of the console.

In the above code, the filemode is set to w, which means the log file is opened in "**write mode**" each time basicConfig() is called, and after each run of the program, it will rewrite the file.

The default configuration for filemode is a, that is **append**, which means that logs will be appended to the log file and adding logs to the existing logs.

Python Logging - Store Logs in a File

There are some basic steps and these are given below:

1. First of all, simply import the logging module just by writing import

logging.

2. The second step is to create and configure the logger. To configure logger to store logs in a file, it is mandatory to pass the **name of the file** in which you want to record the events.
3. In the third step, the format of the logger can also be set. Note that by default, the file works in **append** mode but we can change that to **write** mode if required.
4. You can also set the level of the logger.

So let's move on to the code now:

```
#importing the module

import logging

#now we will Create and configure logger

logging.basicConfig(filename="std.log", format='%(asctime)s %(message)s', filemode='w')

#Let us Create an object

logger=logging.getLogger()

#Now we are going to Set the threshold of logger to DEBUG

logger.setLevel(logging.DEBUG)

#some messages to test

logger.debug("This is just a harmless debug message")

logger.info("This is just an information for you")

logger.warning("OOPS!!!Its a Warning")

logger.error("Have you try to divide a number by zero")

logger.critical("The Internet is not working....")
```

The above code will write some messages to file named **std.log**. If we will open the file then the messages will be written as follows:

```
2020-06-19 12:48:00,449 - This is just harmless debug message
2020-06-19 12:48:00,449 - This is just an information for you
2020-06-19 12:48:00,449 - OOPS!!!Its a Warning
2020-06-19 12:48:00,449 - Have you try to divide a number by zero
2020-06-19 12:48:00,449 - The Internet is not working...
```

You can change the format of logs, log level or any other attribute of the LogRecord along with setting the filename to store logs in a file along with the mode.

3. Python Logging Variable Data

Sometimes you would want to include some **dynamic information** from your application in the logs.

As we have seen in examples till now, we print strings in logs, and we can easily format and create strings by adding variable values to it. This can also be done within the logging methods like debug(), warning(), etc.

To log the variable data, we can use a **string to describe the event** and then **append the variable data as arguments**. Let us see how we will do this with the help of an example given below:

Example

```
import logging


logging.warning('%s before you %s', 'Think', 'speak!')
```

```
WARNING:root: Think before you speak!
```

In the above code, there is merging of **variable data into the event description message** using the old, %s style of string formatting.
Also, the arguments that are passed to the method would be included as **variable data in the message**.

There is Another way

As you can use any formatting style, the **f-strings introduced in Python 3.6** are also an amazing way to format strings as with their help the formatting becomes short and easy to read:
Here is a code example:

```
import logging

name = 'Thomas'


logging.error(f'{name} raised an error in his code')
```

```
ERROR:root: Thomas raised an error in his code
```

4. Python Logging Classes and Functions

In our previous tutorials, we have already seen how we can use and configure the **default logger** named root, which is mainly used **by the logging module** whenever **we call its functions** directly like this: logging.info().

Object	**Description**

Logger.info(msg)	This function mainly helps to log a message with level INFO on thi logger.
Logger.warning(msg)	This function mainly helps to log a message with level WARNING this logger.
Logger.error(msg)	This function mainly helps to log a message with level ERROR on logger.
Logger.critical(msg)	This function mainly helps to log a message with level CRITICAL this logger.
Logger.setLevel(lvl)	This function is mainly used to set the threshold of the logger to lvl that indicates all the messages below this level will be ignored.
Logger.exception(msg)	This function mainly helps to log a message with level ERROR on logger.
Logger.log(lvl, msg)	This function will Logs a message with integer level lvl on this logg
Logger.filter(record)	This method is mainly used to apply the logger's filter to the provid record and it will return True if the record is to be processed. otherwise, it will return False.
Logger.addFilter(filt)	It is used to adds a specific filter filt to this logger.
Logger.removeFilter(filt)	It is used to adds a specific filter filt from this logger.
Logger.hasHandlers()	This is mainly used to check if the logger **has any handler configu or not**.
Logger.addHandler(hdlr)	In order to add a specific handler hdlr to this logger.

Logger.removeHandler(hdlr)	In order to remove a specific handler hdlr to this logger.

You can also define **your own logger by creating an object of the** Logger class, especially in the case if your **application has multiple modules.**

Let us look at some of the classes and functions in the logging module.

The most commonly used classes which are already defined in the logging module are given below:

1. Logger class

The Logger is the class whose objects will be used in the application code directly to call the functions.

2. LogRecord

The Logger class automatically create LogRecord objects which contain the information related to the event being logged, or in simple words the log message. It contains information like the **name of the logger**, the **function**, the **line number**, the message, the process id, etc.

3. Handler

The Handlers are mainly used to send the LogRecord to the required output destination, it **can be console or a file**

The class Handler is the **base for subclasses** like StreamHandler, FileHandler, SMTPHandler, HTTPHandler, and more. These subclasses also **send the logging outputs to corresponding destinations**, like sys.stdout or a disk file.

4. Formatter

The Formatter is a class **where you specify the format of the output** by **specifying a string format** that mainly lists out the attributes the output should contain.

Several Logger objects

In Logging Module items with all caps are constant, the capitalize items indicate classes, and the items which start with lowercase letters are methods. Below we have a table consisting of several logger objects offered by the Logging module:

Chapter 9: Python With MySQL

1. MySQL with Python

In this tutorial, we will learn **What is MySQL**, What are the **prerequisites to use MySQL in Python** and from where you can **download and install MySQL** and **MySQL Python Connector**.

First of all, let us see what are the Prerequisites needed to learn MySQL in Python:

Python MySQL - Prerequisites

Below are the prerequisites needed in order to gain complete understanding:

- Python **Variable**, **Data types**, **Control Structures**, **Loops**, etc. Anyone who wants to put their step into this tutorial must be aware of these topics.
- **Basics of SQL**. If you want to learn the basics of SQL then here is the link for our free course: Learn SQL.

Note:

In this tutorial our main aim to teach you how you can work with MySQL in Python.

What is MySQL?

Let us first learn what MySQL is:

MySQL is **an open-source**, **relational database management system(RDBMS)** that is based on **Structured Query Language**(SQL). One important thing to note here is that MySQL is used to store data and it is not used to create programs; Thus it is not a Programming Language.

- Thus **SQL** can be used to **program a MySQL Database**.
- The main advantage of MySQL is that it can run on any of the Operating System.
- It is one of the **Popular Database**.

If you want to practice the code examples in our upcoming tutorials then it is

mandatory for you to install MySQL on your Computer. It is freely available you can download and install the MySQL database from its official website :

https://www.mysql.com/downloads/

After installing **MySQL** the next step is to install **mysql connector for the python.** Now the question arises what is MySQL connector?

MySQL Connector

Basically **Python needs a MySQL Driver** which is used **to access the MySQL Database** an in this case **"MySQL Connector"** is the Driver

MySQL Connector for Python **is a connector** that enables the **Python programs** to access the **MySQL database**.

There are two ways to download the MySQL Connector:

1. The first way is **you can download and install it** from the link given: https://dev.mysql.com/downloads/connector/python/
2. Another way is **By Using PIP to install the "MySQL Connector"** as PIP is already installed in **your Python environment**. So you can download and install just by **navigating your command line** to the location of PIP and By writing:

```
python -m pip install mysql-connector-python
```

We Recommend **you to use the second way that is by PIP**; also it is your choice.

Now, After this **MySQL Driver that is "MySQL Connector"** is downloaded and installed successfully in your Computer. But in order to confirm, we can Test the MySQL Connector.

Test the MySQL Connector

To check if the MySQL connector is installed correctly or not in your System you just need to write the following code:

```
import mysql.connector
```

If the above code runs successfully, without raising any error then it means installation is done correctly.

Now **we will create the connection** with the help of which further we can create the database, insert into it, and perform other queries too.

Creating the Connection

It is the first step that is **creating the connection.**

Now we will connect to the database using the **username** and the **password** of **MySQL**. If you forgot your **username** or **password**, create a new **user** with a password.

```
###Connecting to the database

###Let us import mysql.connector

import mysql.connector



## Now connecting to the database using 'connect()' method

## it takes 3 required parameters 'host', 'user', 'passwd'



mydb = mysql.connector.connect(

  host="localhost",

  user="yourusername",

  password="yourpassword"

)



print(mydb)
```

It will produce the Output as follows:

```
<mysql.connector.connection_cext.CMySQLConnection object at 0x0000023F50726518>
```

The above output indicates that you have connected to the **MySQL Database**. In our further tutorial, we will start querying the **database with the SQL statements**.

2. Python MySQL - Create Database

We will learn how to **create a database** or how to **check if any database** exists already in MySQL using Python.

To create the database in **MySQL** we will use the "CREATE DATABASE" statement.

Python MySQL - CREATE DATABASE

Below we have the **basic syntax** of the create database statement:

```
CREATE DATABASE database_name
```

Now we will create a database named **"studytonight"**. Let us see how we will create it:

Python MySQL - Create Database Example

Let us create a database named "**studytonight**".Given below is the code snippet for the same:

```
#for our convenience we will import mysql.connector as mysql

import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword"
```

```
)

## Now Let us create an instance of 'cursor' class which is used to execute the 'SQL' statements in 'Python'

cursor = db.cursor()

## creating a database called 'studytonight'

## 'execute()' method is used to compile a 'SQL' statement

## below statement is used to create tha 'studytonight' database

cursor.execute("CREATE DATABASE studytonight")
```

In the case, if the database named "studytonight" **already exists** then the above code will raise an error.

If **there is no error** it means the database is **created successfully**.

Now let us see **how we can check all the databases in our system.**

Python MySQL - List all Database

We can list down all the databases in MySQL using Python and in that list we can also check if any particular Database is available or not.

To check if any database already exists SHOW DATABASES SQL statement is used.

Let us see how:

To list out all the databases in the system.Below is the code snippet for the same:

```
#for our convenience we will import mysql.connector as mysql

import mysql.connector as mysql
```

```
db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword"
)

cursor = db.cursor()

cursor.execute("SHOW DATABASES")

for x in cursor:
  print(x)
```

The output of the above code will list out all the databases in the system:

```
('admin_panel',)
('information_schema',)
('mysql',)
('performance_schema',)
('sakila',)
('studytonight',)
('sys',)
('world',)
('xyz',)
```

As you can see in the code above, we have used Python for loop to iterate over the cursor object and print the names of all the databases found. Similarly, we can compare the name of databases to check if any particular database already exists, before creating it using Python if condition.

So in this tutorial we learned how to create a new database in MySQL using Python and how to list down all the MySQL databases in Python and print them in console.

3. Python MySQL - Create and List Table

Now we will learn **how to create tables in any MySQL** database in Python and we will also see **how to check if a table already exists** or not by listing down all the tables in a database using Python.

Python MySQL - Create Table

Basically, to **store information in the MySQL database** there is a need to create the tables. It is also **required to select our database first** and then create tables inside that database.

At the time of creating the connection, you can also specify the name of your database, like given below:

```
import mysql.connector

db = mysql.connector.connect(
  host = "localhost",
  user = "yourusername",
  password = "yourpassword",
```

```
  database = "studytonight"

)
```

If the above code is **executed without any errors** then it means you have successfully connected to the database named **studytonight**.

SQL Query to Create Table

To create a table in the selected database the following statement will be used. Let us see the syntax:

```
CREATE TABLE table_name;
```

Let us create a table named **students** in the specified database, that is **studytonight**

In the table **students** we will have the following fields: **name**, **rollno**, **branch**, and **address**.

```
#for our convenience we will import mysql.connector as mysql

import mysql.connector as mysql


db = mysql.connect(

  host = "localhost",

  user = "yourusername",

  passwd = "yourpassword",

  database="studytonight"

)


cursor = db.cursor()
```

```
cursor.execute("CREATE TABLE students (name VARCHAR(255), rollno INTEGER(100), branch VARCHAR(255), address VARCHAR(255))")
```

If this code executes **without any error** then it means the table has been created successfully.

Now, If you want to **check the existing tables** in the database then you can use the SHOW TABLES SQL statement.

List existing Tables in a Database

Now as we have created a table in our database. Let us check the tables that exist in our database. Use the code given below:

```
#for our convenience we will import mysql.connector as mysql

import mysql.connector as mysql


db = mysql.connect(

    host = "localhost",

    user = "yourusername",

    passwd = "yourpassword",

    database="studytonight"

)


cursor = db.cursor()

## getting all the tables which are present in 'datacamp' database

cursor.execute("SHOW TABLES")


tables = cursor.fetchall() ## it returns list of tables present in the database
```

```
## showing all the tables one by one
for table in tables:
    print(table)
```

The output of the above code is as follows

```
('students',)
```

Python MySQL - Table with Primary Key

As we had created a table named **students** in our database, in which we will store student data and fetch it whenever required. But while fetching data, we might **find students with same name** and that can lead to wrong data getting fetched, or cause some confusion.

So to **uniquely identify each record** in a table we can use **Primary Key** in our tables. Let us see first **what is a Primary key?**

What is Primary Key?

A primary key is an attribute to make a column or a set of columns accept only **unique values**. With the help of the primary key, one can **find each row uniquely in the table.**

Watch this video to learn about DBMS Keys - DBMS Keys Explained with Examples

Thus in order to identify each **row uniquely with a number starting from 1**. We will use the syntax as follows:

```
INT AUTO_INCREMENT PRIMARY KEY
```

Using the above code with any column, we can make its value as auto increment, which means the database will automatically add an incremented value even if you do not insert any value for that column while inserting a

new row of data to your table.

Add Primary Key during Table creation

Let us see **how to add a primary key** at the time of table creation. The code snippet for the same is given below:

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)
cursor = db.cursor()

## creating the 'students' table with the 'PRIMARY KEY'

cursor.execute("CREATE TABLE students (name VARCHAR(255), rollno INTEGER(100) NOT NULL AUTO_INCREMENT PRIMARY KEY, branch VARCHAR(255), address VARCHAR(255))")
```

If the above code runs without an error then it means you have successfully created a table named "students" with the column **rollno** as primary key.

Python MySQL - Describe the Table

We can use the below python code to describe any table to see what all columns it has and all the meta information about the table and all its columns.

```
import mysql.connector as mysql

db = mysql.connect(
    host = "localhost",
    user = "yourusername",
    passwd = "yourpassword",
    database = "studytonight"
)

cursor = db.cursor()

cursor.execute("DESC students")

print(cursor.fetchall())
```

The output will be:

```
[('name', 'varchar(255)', 'YES', '', None, ''), ('rollno', 'int', 'NO', 'PRI', None, 'auto_increment'), ('branch', 'varchar(255)', 'YES', '', None, ''), ('address', 'varchar(255)', 'YES', '', None, '')]
```

Add Primary Key to Existing Table

Now in this example, we will assume that **rollno** column does not exist in our **student** table. So we will learn how to **add a column** to be **used as primary key** in an existing table.

Let us see how to create a primary key on an existing table. The code snippet

for the same is given below:

```
import mysql.connector as mysql

db = mysql.connect(
    host = "localhost",
    user = "root",
    passwd = "himaniroot@99",
    database = "studytonight"
)

cursor = db.cursor()

## We are going to add rollno field with primary key in table students

cursor.execute("ALTER TABLE students ADD COLUMN rollno INT AUTO_INCREMENT PRIMARY KEY")

print(cursor.fetchall())
```

In the above code we have used the SQL Alter Query to add a new column to an existing table. You can use the describe table query to see the new added column.

```
[('name', 'varchar(255)', 'YES', '', None, ''), ('branch', 'varchar(255)', 'YES', '', None, ''), ('address', 'varchar(255)', 'YES', '', None, ''), ('rollno', 'int', 'NO', 'PRI', None, 'auto_increment')]
```

So in this tutorial we have covered everything related to creating a new MySQL table in python. We covered how to alter a table, how to list down all

the tables in a database, and how to define a column as a primary key.

4. Python MySQL - Insert data in Table

We will learn how to **insert a single row** and **insert multiple rows** of data in a MySQL table using Python.

To **Insert data into a MySQL Table** or to add data to the MySQL Table which we have created in our previous tutorial, We will use the INSERT SQL statement.

If you are new to SQL, you should first learn about the SQL INSERT statement.

Python MySQL - INSERT Data

Let us see the **basic syntax** to use INSERT INTO statement to insert data into our table:

```
INSERT INTO table_name (column_names) VALUES(data)
```

We can insert data in a table in two ways:

- Inserting a **single row** at a time
- Inserting **multiple rows** at a time

Inserting Single Row in MySQL Table

In this section, we will see the code example for inserting a single row of data in a MySQL table:

We will be adding data to the **students** table we created in Python MySQL - Create Table tutorial.

```
import mysql.connector as mysql
```

```
db = mysql.connect(
    host = "localhost",
    user = "root",
    passwd = "himaniroot@99",
    database = "studytonight"
)

cursor = db.cursor()
## defining the Query
query ="INSERT INTO students(name, branch, address) VALUES (%s, %s,%s)"
## There is no need to insert the value of rollno
## because in our table rollno is autoincremented #started from 1
## storing values in a variable
values = ("Sherlock", "CSE", "220 Baker Street, London")

## executing the query with values
cursor.execute(query, values)

## to make final output we have to run
## the 'commit()' method of the database object
db.commit()

print(cursor.rowcount, "record inserted")
```

The output of the above code will be:

```
1 record inserted
```

Inserting Multiple Rows in MySQL Table

In this section, we will see the code example for inserting multiple rows of data in a MySQL table.

To insert multiple rows into the table, the executemany() method is used. It takes a list of **tuples** containing the data as a second parameter and the **query** as the first argument.

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "root",
  passwd = "himaniroot@99",
  database = "studytonight"
)

cursor = db.cursor()
## defining the Query
query ="INSERT INTO students(Name, Branch,Address) VALUES (%s, %s, %s)"

## storing values in a variable
values = [
  ("Peter", "ME","Noida"),
  ("Amy", "CE","New Delhi"),
```

```
    ("Michael", "CSE","London")

]

## executing the query with values

cursor.executemany(query, values)

## to make final output we have to run

## the 'commit()' method of the database object

db.commit()

print(cursor.rowcount, "records inserted")
```

If all three rows were entered successfully, the output of the above code will be:

```
3 records inserted
```

So in this tutorial we learned how we can insert data into MySQL table in Python.

5. Python MySQL - Select data from Table

We will learn **how to retrieve data from MySQL table** in python, both, the complete table data, and data from some specific columns.

Python MySQL - SELECT Data

In MySQL, to **retrieve data from a table** we will use the SELECT statement. The syntax for the same is given below:

```
SELECT column_names FROM table_name
```

Retrieve All records from MySQL Table

In order to get all the records from a table, * is used instead of column names. Let us retrieve all the data from the **students** table which we inserted before:

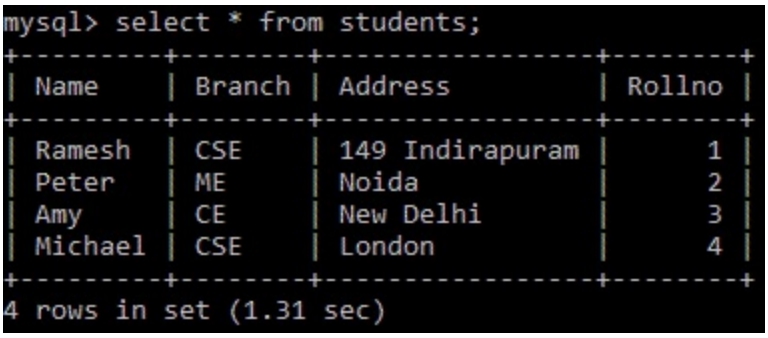
```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)

cursor = db.cursor()

## defining the Query
query = "SELECT * FROM students"

## getting records from the table
cursor.execute(query)
```

```
## fetching all records from the 'cursor' object

records = cursor.fetchall()


## Showing the data

for record in records:

   print(record)
```

Thus the output of the above code will be:

```
('Ramesh', 'CSE', '149 Indirapuram', 1)

('Peter', 'ME', 'Noida', 2)

('Amy', 'CE', 'New Delhi', 3)

('Michael', 'CSE', 'London', 4)
```

Below we have a snapshot of the exact output when we run this python code:



In the next section we will learn how to retrieve data of certain columns from a table.

Retrieve data from specific Column(s) of a Table

In order to select data from some columns of the table just mention the column name after the **SELECT** in the syntax mentioned above:

```
import mysql.connector as mysql
```

```
db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)

cursor = db.cursor()

## defining the Query
query = "SELECT name FROM students"

## getting 'name' column from the table
cursor.execute(query)

## fetching all usernames from the 'cursor' object
names = cursor.fetchall()

## Showing the data
for name in names:
  print(name)
```

The above code will fetch the **name** column from the **students** table:

```
('Ramesh',)
('Peter',)
```

('Amy',)

('Michael',)

Selecting Multiple columns from a Table

You can also **select multiple columns from a table** at a time by providing multiple column names in the above syntax. Let us see the code snippet given below for clear understanding:

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)

cursor = db.cursor()
## defining the Query
query = "SELECT name, branch FROM students"

## getting 'name', 'branch' columns from the table
cursor.execute(query)

## fetching all records from the 'cursor' object
data = cursor.fetchall()
```

```
## Showing the data

for pair in data:

    print(pair)
```

The above code will fetch both **name** and **branch** column both from the table **students**:

```
('Ramesh', 'CSE')

('Peter', 'ME')

('Amy', 'CE')

('Michael', 'CSE')
```

To fetch the first record - fetchone()

In the above examples, we saw that **all rows are fetched** because we were using fetchall() method. Now to fetch only a single-row fetchone() method will be used. This method will return the first row from the records fetched by the query. Let us see the code snippet given below:

```
import mysql.connector as mysql

db = mysql.connect(

    host = "localhost",

    user = "yourusername",

    passwd = "yourpassword",

    database = "studytonight"

)



cursor = db.cursor()

cursor.execute("SELECT * FROM students")

```

```
myresult = cursor.fetchone() ##fetches first row of the record

print(myresult)
```

Thus in the output the first row of the record will be fetched:

```
('Ramesh', 'CSE', '149 Indirapuram', 1)
```

So in this tutorial, we learned various ways to retrieve data from a MySQL table in Python.

5. Python MySQL - Update Table data

In this tutorial, we will learn how to **Update MySQL table data** in Python where we will be using the UPDATE SQL query and the WHERE clause.

The UPDATE SQL query is **used to update any record** in MySQL table.

Python MySQL UPDATE: Syntax

Below we have the basic syntax of the UPDATE statement:

```
UPDATE table_name SET column_name = new_value WHERE condition
```

The above syntax is used to update any **specific row** in the MySQL table. And to specify which specific row of data we want to update, we use the WHERE clause to provide the condition to be matched while looking for the right row of data to update.

Python MySQL UPDATE Table Data: Example

Let us update the record in the **students** table (from the Python MySQL

create table tutorial) by changing the **name** of the student whose **rollno** is **3**. The code is given below:

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)
cursor = db.cursor()
## defining the Query
query = "UPDATE students SET name = 'Navin' WHERE rollno = 3"
## executing the query
cursor.execute(query)
## final step is to commit to indicate the database
## that we have changed the table data
db.commit()
```

To check if the **data is updated successfully** we can retrieve the students table data using the given below code:

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
```

```
    user = "yourusername",
    passwd = "yourpassword",
    database = "studytonight"
)

cursor = db.cursor()
## defining the Query
query = "SELECT * FROM students"
## getting records from the table
cursor.execute(query)
## fetching all records from the 'cursor' object
records = cursor.fetchall()
## Showing the data
for record in records:
    print(record)
```

The output of the above code is:

```
('Ramesh', 'CSE', '149 Indirapuram', 1)
('Peter', 'ME', 'Noida', 2)
('Navin', 'CE', 'New Delhi', 3)
```

Here is the snapshot of the actual output:

```
mysql> UPDATE students SET name = 'Navin' WHERE rollno = 3;
Query OK, 0 rows affected (0.06 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql> select * from students;
+--------+--------+-----------------+--------+
| Name   | Branch | Address         | Rollno |
+--------+--------+-----------------+--------+
| Ramesh | CSE    | 149 Indirapuram |      1 |
| Peter  | ME     | Noida           |      2 |
| Navin  | CE     | New Delhi       |      3 |
+--------+--------+-----------------+--------+
3 rows in set (0.00 sec)
```

In the SELECT query too, we can use the WHERE clause to retrieve only the data of the row with **rollno** 3 or any other condition.

7. Python MySQL - Delete Table Data

We will learn how to **DELETE MySQL table data** in Python where we will be using the DELETE SQL query and the WHERE clause.

The DELETE SQL query is **used to delete any record** from MySQL table.

Python MySQL DELETE: Syntax

The syntax for the same is given below:

```
DELETE FROM table_name WHERE condition
```

Note: In the above syntax, WHERE clause is used to specify the condition to pin point a particular record that you want to delete. If you don't specify the condition, then all of the records of the table will be deleted.

Python MySQL DELETE Table Data: Example

Let us see an example to delete the record in the **students** table (from the Python MySQL create table tutorial) whose **rollno** is **4**. The code is given below:

```
import mysql.connector as mysql

db = mysql.connect(
    host = "localhost",
    user = "yoursername",
    passwd = "yourpassword",
    database = "studytonight"
)

cursor = db.cursor()
## defining the Query
query = "DELETE FROM students WHERE Rollno = 4"

## executing the query
cursor.execute(query)

## final step to tell the database that we have changed the table data
db.commit()
```

If the above code runs without an error then it means that the row with **rollno = 4** is deleted successfully.

To check if it exists in the table or not you can use the code given below:

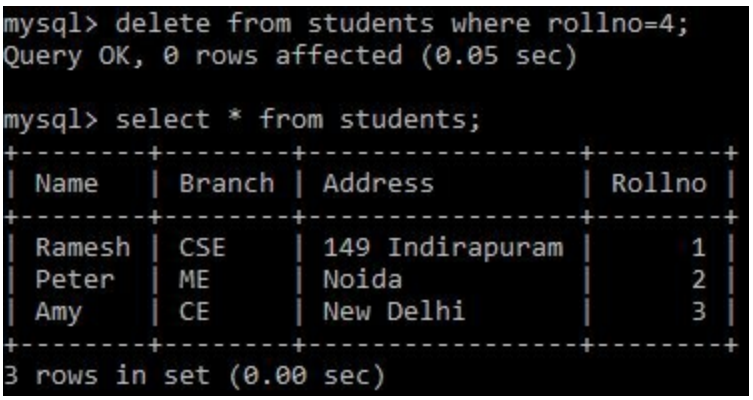
```
import mysql.connector as mysql

db = mysql.connect(
```

```
    host = "localhost",
    user = "yourusername",
    passwd = "yourpassword",
    database = "studytonight"
)

cursor = db.cursor()
## defining the Query
query = "SELECT * FROM students"

## getting records from the table
cursor.execute(query)

## fetching all records from the 'cursor' object
records = cursor.fetchall()

## Showing the data
for record in records:
    print(record)
```

The output for the above code is given below:

```
('Ramesh', 'CSE', '149 Indirapuram', 1)
('Peter', 'ME', 'Noida', 2)
('Amy', 'CE', 'New Delhi', 3)
```

Here is a snapshot of the actual output:



```
mysql> delete from students where rollno=4;
Query OK, 0 rows affected (0.05 sec)

mysql> select * from students;
+--------+--------+-----------------+--------+
| Name   | Branch | Address         | Rollno |
+--------+--------+-----------------+--------+
| Ramesh | CSE    | 149 Indirapuram |      1 |
| Peter  | ME     | Noida           |      2 |
| Amy    | CE     | New Delhi       |      3 |
+--------+--------+-----------------+--------+
3 rows in set (0.00 sec)
```

As we can see in the output above, the row with **rollno=4** is successfully deleted.

3. Python MySQL - Drop Table

We will learn **how to delete a MySQL table** or drop the table completely from the database using Python.

To completely **delete an existing table**(both table data and table itself), the DROP TABLE SQL statement is used.

Python MySQL DROP TABLE: Example

Let's assume we have a table with name **customers** in the **studytonight** database. Then we can use the below code to drop that table:

```
import mysql.connector as mysql


db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
```

```
)

cursor = db.cursor()
## We have created another table in our database named customers
## and now we are deleting it
sql = "DROP TABLE customers"

cursor.execute(sql)
```

If the above code will execute without any error it means table named **customers** is deleted succesfully.

Python MySQL - Drop Table if it exists

The IF EXISTS keyword is used to avoid error which may occur if you try to drop a table which doesn't exist.

When we use the IF EXISTS clause, we are informing the SQL engine that if the given table name exists, then drop it, if it doesn't exists, then do nothing.

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)
```

```
cursor = db.cursor()

sql = "DROP TABLE IF EXISTS customers"

cursor.execute(sql)
```

If the code executes without an error, then it means the customers table is deleted if it existed.

Here is the snapshot of the actual output:

```
mysql> DROP TABLE IF EXISTS customers;
Query OK, 0 rows affected, 1 warning (0.38 sec)
```

So in this tutorial we learned how to drop a MySQL table using Python. This is useful when we have some application which creates some temporary tables to store some data and then after the processing, deletes those tables.

). Python MySQL - WHERE Clause

We will learn **how to filter rows** from the fetched resultset, or update a specific row, or delete a specific row from a MySQL table using the WHERE clause to specify the condition to find the record/row of data on which the operation is performed.

So, WHERE clause is nothing but a way to provide a condition(or multiple conditions) to the SQL engine, which is applied on the query resultset to filter out the required records of data.

Python MySQL WHERE Clause

We have already used the WHERE clause in our previous tutorials:

- Python MySQL - Update Table data

- Python MySQL - Delete Table data

If you want to **select data from a table based on some condition** then you can use WHERE clause in the SELECT statement.

- The WHERE clause is mainly used for **filtering the rows from the result set**.
- It is **helpful** in **fetching, updating, and deleting** data from the MySQL Table.

The syntax of using WHERE clause in SELECT statement is as follows:

```
SELECT column_name

FROM table_name

WHERE condition;
```

The above syntax is useful in fetching the records on the basis of some conditions.

Using WHERE Clause

Below we have an example where we will fetch the row having **rollno = 2** from our **students** table(from the Python MySQL create table tutorial):

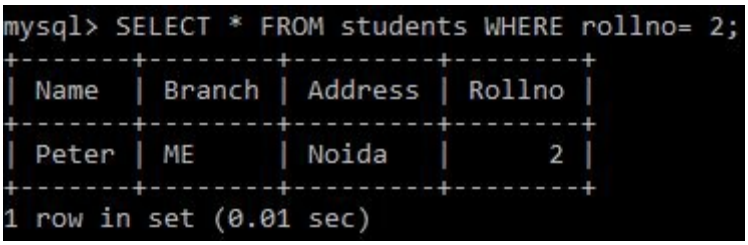
```
import mysql.connector as mysql

###First create a connection between mysql and python

db = mysql.connect(

   host = "localhost",

   user = "yourusername",

   passwd = "yourpassword",

   database = "studytonight"

)

# now create a cursor object on the connection object
```

```
# created above by using cursor() method
cursor = db.cursor()

## defining the Query
query = "SELECT * FROM students WHERE rollno= 2"

## getting records from the table
cursor.execute(query)

## fetching all records from the 'cursor' object
records = cursor.fetchall()

## Showing the data
for record in records:
   print(record)
```

The output of the above code will be:

```
('Peter', 'ME', 'Noida', 2)
```

Here is the snapshot of the actual output:



10. Python MySQL - Orderby Clause

We will learn **how to sort the result in any order**(ascending or descending) in MySQL.

The ORDER BY in MySQL is mainly used **for the sorting purpose**. That is with the help of ORDER BY one can sort the result either in **Ascending** or **Descending** Order.

- By default ORDER BY statement will sort the result in the **Ascending order**, or we can also use the ASC keyword.
- In order to sort the result in the **Descending order**, the DESC keyword will be used.

Below we have a **basic syntax** to sort the result in ascending order which is the default:

```
SELECT column_names FROM table_name ORDER BY column_name
```

Python MySQL ORDER BY Example

Below we have an example in which we will sort the result in ascending order. Let us see the code snippet given below:

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)

cursor = db.cursor()
## defining the Query
```

```
query = "SELECT * FROM students ORDER BY name"

## getting records from the table
cursor.execute(query)

## fetching all records from the 'cursor' object
records = cursor.fetchall()

## Showing the data
for record in records:
    print(record)
```

The above code will sort the names in ascending order and the output will be as follows:

```
('Amy', 'CE', 'New Delhi', 3)
('Michael', 'CSE', 'London', 4)
('Peter', 'ME', 'Noida', 2)
('Ramesh', 'CSE', '149 Indirapuram', 1)
```

Here is the snapshot of the actual output:

```
mysql> Select * From students ORDER BY name;
+---------+--------+-----------------+--------+
| Name    | Branch | Address         | Rollno |
+---------+--------+-----------------+--------+
| Amy     | CE     | New Delhi       |      3 |
| Michael | CSE    | London          |      4 |
| Peter   | ME     | Noida           |      2 |
| Ramesh  | CSE    | 149 Indirapuram |      1 |
+---------+--------+-----------------+--------+
4 rows in set (0.21 sec)
```

Python MySQL ORDER BY DESC

The syntax ORDER BY COLUMN_NAME DESC statement is used to sort the resultset based on the specified column in descending order.

The syntax to use this statement is given below:

```
SELECT column_names FROM table_name ORDER BY column_name DESC
```

Now Let us sort the data in **descending order** by **name** column using the DESC keyword with the ORDER BY clause. Let's see the code which is given below:

```
import mysql.connector as mysql

db = mysql.connect(
   host = "localhost",
   user = "yourusername",
   passwd = "yourpassword",
   database = "studytonight"
)

cursor = db.cursor()
## defining the Query
query = "SELECT * FROM students ORDER BY name Desc"

## getting records from the table
cursor.execute(query)

## fetching all records from the 'cursor' object
records = cursor.fetchall()
```

```
## Showing the data
for record in records:
   print(record)
```

The output of the above code will be:

```
('Ramesh', 'CSE', '149 Indirapuram', 1)
('Peter', 'ME', 'Noida', 2)
('Michael', 'CSE', 'London', 4)
('Amy', 'CE', 'New Delhi', 3)
```

Here is the snapshot of the actual output:

```
mysql> Select * From students ORDER BY name Desc;
+---------+--------+-----------------+--------+
| Name    | Branch | Address         | Rollno |
+---------+--------+-----------------+--------+
| Ramesh  | CSE    | 149 Indirapuram |      1 |
| Peter   | ME     | Noida           |      2 |
| Michael | CSE    | London          |      4 |
| Amy     | CE     | New Delhi       |      3 |
+---------+--------+-----------------+--------+
4 rows in set (0.11 sec)
```

11. Python MySQL - Limit Clause

We will learn **how to limit the number of rows** returned in a resultset with the help of LIMIT clause added to the query in the python code.

The syntax to use this clause is given below:

```
SELECT {fieldname(s) | *} FROM tableName(s) [WHERE condition] LIMIT  N;
```

The above syntax indicates the following things:

- **SELECT {fieldname(s) | *} FROM tableName(s)**: This part is used to select the records that we would like to return in our query.

- **[WHERE condition]**: The WHERE clause is optional, but if used then it applies the filter on the result set.
- **LIMIT N**: It is used to limit the records in the result. Here **N** starts from 0 but if you will pass LIMIT 0(then it does not return any record). If you will pass **6** then it will return the starting 6 rows in the output. Suppose the records in the specified table are less than N, then all the records from the table are returned in the result set.

Python MySQL LIMIT: Example

Let us select only two rows from the **students** table(from the Python MySQL create table tutorial). The code for the same is given below:

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)

cursor = db.cursor()
cursor.execute("SELECT * FROM students LIMIT 2")

myresult = cursor.fetchall()

for x in myresult:
 print(x)
```

The output of the above code will be as follows:

```
('Ramesh', 'CSE', '149 Indirapuram', 1)
('Peter', 'ME', 'Noida', 2)
```

Using OFFSET Keyword with LIMIT clause

If you do not want to start from the first position then by using OFFSET keyword in the LIMIT query you can start from any other position. Let us see the code example for the same:

```
import mysql.connector as mysql

db = mysql.connect(
    host = "localhost",
    user = "yourusername",
    passwd = "yourpassword",
    database = "studytonight"
)

cursor = db.cursor()

cursor.execute("SELECT * FROM students LIMIT 2 OFFSET 1")

myresult = cursor.fetchall()

for x in myresult:
```

```
print(x)
```

In the above code **LIMIT 2** means it will return the 2 rows and **OFFSET 1** the resultset will start from the 1st row, means only row 2 is returned. Thus the output is as follows:

```
('Peter', 'ME', 'Noida', 2)
('Navin', 'CE', 'New Delhi', 3)
```

Here is the snapshot of the actual output:

```
mysql> select * from students LIMIT 2;
+--------+--------+-----------------+--------+
| Name   | Branch | Address         | Rollno |
+--------+--------+-----------------+--------+
| Ramesh | CSE    | 149 Indirapuram |      1 |
| Peter  | ME     | Noida           |      2 |
+--------+--------+-----------------+--------+
2 rows in set (0.02 sec)

mysql> select * from students LIMIT 2 offset 1;
+-------+--------+-----------+--------+
| Name  | Branch | Address   | Rollno |
+-------+--------+-----------+--------+
| Peter | ME     | Noida     |      2 |
| Navin | CE     | New Delhi |      3 |
+-------+--------+-----------+--------+
2 rows in set (0.00 sec)
```

The OFFSET keyword isused to specify the starting point. For example if a query returnd 100 rows of data and we have specifed the OFFSET as 50, then data starting from 51st row till the 100th row will be returned.

12. Python MySQL - Table Joins

We will learn how to join two or more MySQL tables in Python.

In order to **combine two or more tables** in MySQL, JOIN statement is used. One thing to note here is that there must be a **common column in both**

tables based on which this operation will be performed.

We have already created the **students** table in the **studytonight** database(from the Python MySQL create table tutorial). Let us create another table named **results** after that we will join both the tables.

Below we have code to create a table named results.
Both **results** and **students** table have one field common that is **rollno**.

```
mysql> create table Results (Rollno int(100) primary key not null, Status varchar(20) not null);
Query OK, 0 rows affected, 1 warning (4.82 sec)

mysql> insert into Results values(1,"Pass");
Query OK, 1 row affected (0.50 sec)

mysql> insert into Results values(2,"Fail");
Query OK, 1 row affected (0.24 sec)

mysql> insert into Results values(3,"Pass");
Query OK, 1 row affected (0.13 sec)

mysql> Select * from Results
    -> Select * from Results;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version fo
 Results' at line 2
mysql> select * from Results;
+--------+--------+
| Rollno | Status |
+--------+--------+
|      1 | Pass   |
|      2 | Fail   |
|      3 | Pass   |
+--------+--------+
3 rows in set (0.00 sec)
```

You can directly run the query to create the new table in MySQL command line.

Python MySQL - Joining two tables

Below we have a code snippet where we will **join two tables** named **results** and **students** based on the column **rollno**:

```
import mysql.connector as mysql


db = mysql.connect(
   host = "localhost",
   user = "yourusername",
   passwd = "yourpassword",
   database = "studytonight"
)
```

```
cursor = db.cursor()

sql = "SELECT students.rollno, students.name,students.Branch,students.Address, results.status from students INNER JOIN results ON students.rollno=results.rollno;"

cursor.execute(sql)

myresult = cursor.fetchall()

for x in myresult:
  print(x)
```

The output of the above code will be:

```
(1, 'Ramesh', 'CSE', '149 Indirapuram', 'Pass')
(2, 'Peter', 'ME', 'Noida', 'Fail')
(3, 'Navin', 'CE', 'New Delhi', 'Pass')
```

The above output indicates that both tables are joined.

Here is the snapshot of the actual output:

```
mysql> select students.rollno, students.name,students.Branch,students.Address, results.status from students INNER JOIN results ON students.rollno=results.rollno;
+--------+--------+--------+-----------------+--------+
| rollno | name   | Branch | Address         | status |
+--------+--------+--------+-----------------+--------+
|      1 | Ramesh | CSE    | 149 Indirapuram | Pass   |
|      2 | Peter  | ME     | Noida           | Fail   |
|      3 | Navin  | CE     | New Delhi       | Pass   |
+--------+--------+--------+-----------------+--------+
3 rows in set (0.00 sec)
```

Python MySQL - Left Join

It is important to note that the INNER JOIN(which we covered in the example above) only shows the rows in the resultset **when there is a match**.

If you want to **fetch all the records fromthea left-hand side table** even if

there is **no match** then Left Join will be used.
Let us see an example given below for the Left Join:

```
import mysql.connector as mysql

db = mysql.connect(
  host = "localhost",
  user = "yourusername",
  passwd = "yourpassword",
  database = "studytonight"
)

cursor = db.cursor()

sql = "SELECT students.rollno, students.name,students.Branch,students.Address, results.status from students LEFT JOIN results ON students.rollno=results.rollno;"

cursor.execute(sql)

myresult = cursor.fetchall()

for x in myresult:
  print(x)
```

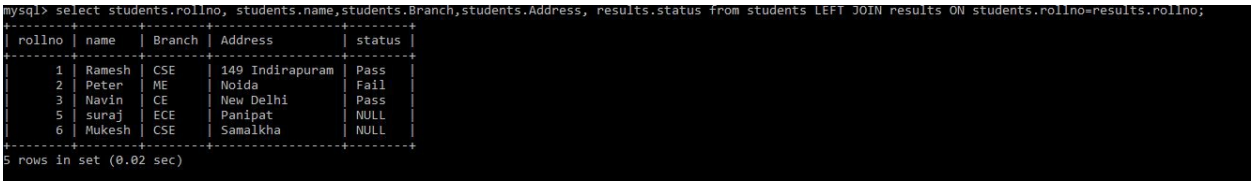
The output of the above code is as follows. Let us see:

```
(1, 'Ramesh', 'CSE', '149 Indirapuram', 'Pass')
(2, 'Peter', 'ME', 'Noida', 'Fail')
```

```
(3, 'Navin', 'CE', 'New Delhi', 'Pass')
(5, 'suraj', 'ECE', 'Panipat', None)
(6, 'Mukesh', 'CSE', 'Samalkha', None)
```

In the above output, the status of **rollno 5 and 6 is None** because their result is **not mentioned in the results table**. But as we have applied LEFT JOIN so the query selects all the rows from the left table **even if there is no match**.

Here is the snapshot of the actual output:



Python MySQL - Right Join

If you want to fetch all the records from the **right-hand side table** even if there is **no match** then Right Join will be used.

Let us see an example given below for the Right Join. The code given below will fetch all the rows from the right-hand side table. It will not return those rows with **rollno 5 and 6**:

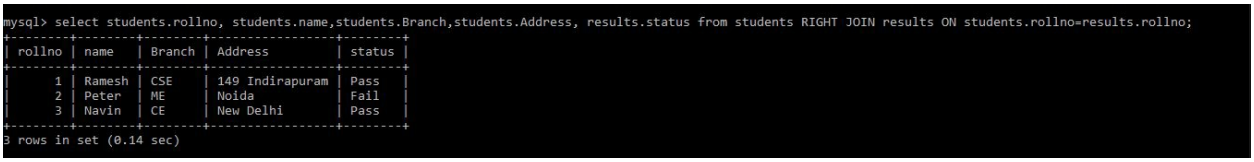
```
import mysql.connector as mysql

db = mysql.connect(
   host = "localhost",
   user = "yourusername",
   passwd = "yourpassword",
   database = "studytonight"
)

cursor = db.cursor()
```

```
sql = "SELECT students.rollno, students.name,students.Branch,students.Address, results.status from students RIGHT JOIN results ON students.rollno=results.rollno;"

cursor.execute(sql)

myresult = cursor.fetchall()

for x in myresult:
  print(x)
```

The output of the above code is as follows:

```
(1, 'Ramesh', 'CSE', '149 Indirapuram', 'Pass')
(2, 'Peter', 'ME', 'Noida', 'Fail')
(3, 'Navin', 'CE', 'New Delhi', 'Pass')
```

Here is the snapshot of the actual output:



And with this we have covered all the basics of Python MySQL. If you have to develop an application in which you want to have a database with multipl tables, in which you want to store data and retrieve data from the tables, then we hope these tutorials help you.

Conclusion

This book is dedicated to the readers who take time to write me each day.

Every morning I’m greeted by various emails — some with requests, a few with complaints, and then there are the very few that just say thank you. All these emails encourage and challenge me as an author — to better both my books and myself.

Thank you!