Programming language (Python)

- Introduction to Python History
- Features, Setting up path Basic Syntax, Comments, Variable
- Different Data Types
- Casting, string, Boolean
- Python Operators
- Conditional Statements
- Looping
- Control Statements, String Manipulation, Lists, Tuple, sets
- Dictionaries
- Arrays
- Iterators, modules, dates, math,
- Modules, Input and Output.
- Functions & arguments
- Modules ● Exception Handling
- Built in Functions in Python.
- File handling in Python.
- Python Architecture.
- Documentation in Python

INTRODUCTION TO PYTHON HISTORY

Python is a popular programming language. It is designed by **Guido van Rossum** in **1991**. Nowadays, Python is one of the most popular and widely used programming language all over the world. It is a high-level programming language. It's syntax allows programmers to express concepts in fewer lines of code. It is used for set of tasks including console based, GUI based, web programming and data analysis. Python is a easy to learn and simple programming language so even if you are new to programming, you can learn python without facing any problems.

Story behind Python's invention

In the late 1980s, working on Python started by Guido Van Rossum. He began doing its application-based work in December of 1989 at Centrum Wiskunde & Informatica (CWI) which is situated in the Netherlands. It was started as a hobby project because he was looking for an interesting project to keep him occupied during Christmas.

From ABC language to Python

The success of Python programming language came from **ABC Programming Language**, which was operating on Amoeba Operating System and had the feature of exception handling. Guido Van Rossum had already helped create ABC language earlier in his career and had seen some issues with ABC language but liked most of the features. While creating Python, he had taken the syntax of ABC language, and some of its good features also. Initially it came with a lot of complaints too, so he fixed those issues completely and created a good scripting language that had removed all the flaws.

Named Python after a TV Show

The inspiration for the name came from the **BBC's TV Show – 'Monty Python's Flying Circus'**, as Guido Van Rossum was a big fan of the TV show and also he wanted a short, unique and slightly mysterious name for his invention and hence he named it Python! For quite some time he worked for Google, but currently, he is working at Dropbox.

Evolution of Python

The language was finally released in 1991. When it was released, it used a lot fewer codes to express the concepts, when we compare it with Java, C++ & C. Its design philosophy was quite good too. Its main objective is to provide code readability and advanced developer productivity. When it was released, it had more than enough capability to provide classes with inheritance, several core data types of exception handling and functions.

FEATURES, SETTING UP PATH BASIC SYNTAX, COMMENTS, VARIABLE

Python is a feature rich high-level, interpreted, interactive and object-oriented scripting language.

Features of Python

There are some important features of Python programming language...

1. Easy to Learn

This is one of the most important reasons for the popularity of Python. Python has a limited set of keywords. Its features such as simple syntax, usage of indentation to avoid clutter of curly brackets and dynamic typing that doesn't necessitate prior declaration of variable help a beginner to learn Python quickly and easily.

2. Interpreter Based

Any Programming languages is either compiler based or interpreter based. Python is an interpreter based language. The interpreter takes one instruction from the source code at a time, translates it into machine code and executes it. Instructions before the first occurrence of error are executed. With this feature, it is easier to debug the program and thus proves useful for the beginner level programmer to gain confidence gradually. Python therefore is a beginner-friendly language.

3. Interactive

Python prompt >>> works on the principle of **REPL** (Read – Evaluate – Print – Loop). You can type any valid Python expression here and press Enter. Python interpreter immediately returns the response and the prompt comes back to read the next expression like this **>>>**.

Example-1:

```
>>> 2*3+1

7
```

Example-2:

```
>>> print ("Hello World")

Hello World
```

The interactive mode is especially useful to get familiar with a library and test out its functionality. You can try out small code snippets in interactive mode before writing a program.

4. MultiParadigm

Python is a completely object-oriented language. Everything in a Python program is an object. However, Python conveniently encapsulates its object orientation to be used as an imperative or procedural language-such as C. Python also provides certain functionality that resembles functional programming. Moreover, certain third-party tools have been developed to support other programming paradigms such as aspect-oriented and logic programming.

5. Standard Library

Even though it has a very few keywords (only Thirty Five), Python software is distributed with a standard library made of large number of modules and packages. Thus Python has out of box support for programming needs such as serialization, data compression, internet data handling, and many more. Python is known for its batteries included approach.

6. Open Source and Cross Platform

Python's standard distribution can be downloaded from https://www.python.org/downloads/ without any restrictions. In addition, the source code is also freely available, which is why it comes under open source category.

Python is a cross-platform language. Pre-compiled binaries are available for use on various operating system platforms such as Windows, Linux, Mac OS, Android OS. A Python program can be easily ported from one OS platform to other.

7. GUI Applications

Python's standard distribution has an excellent graphics library called TKinter. It is a Python port for the vastly popular GUI toolkit called TCL/Tk. You can build attractive user-friendly GUI applications in Python. Examples are PyQt, WxWidgets, PySimpleGUI etc.

8. Database Connectivity

Almost any type of database can be used as a backend with the Python application. DB-API is a set of specifications for database driver software to let Python communicate with a relational database. With many third party libraries, Python can also work with NoSQL databases such as MongoDB.

9. Extensible

The term extensibility implies the ability to add new features or modify existing features. As stated earlier, CPython (which is Python's reference implementation) is written in C. Hence one can easily write modules/libraries in C and incorporate them in the standard library. There are other implementations of Python such as Jython (written in Java) and IPython (written in C#). Hence, it is possible to write and merge new functionality in these implementations with Java and C# respectively.

10. Active Developer Community

As a result of Python's popularity and open-source nature, a large number of Python developers often interact with online forums and conferences. Python Software Foundation also has a significant member base, involved in the organization's mission to "Promote, Protect, and Advance the Python Programming Language". Major IT companies Google, Microsoft, and Meta contribute immensely by preparing documentation and other resources.

More Features of Python

Apart from the above-mentioned features, Python has another big list of good features, few are listed below...

1. It supports functional and structured programming methods as well as OOP.

2. It can be used as a scripting language or can be compiled to byte-code for building large applications.

3. It provides very high-level dynamic data types and supports dynamic type checking.

4. It supports automatic garbage collection.

5. It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

SETTING UP PATH FOR PYTHON

1. Get Python Installer from python.org.

2. Get the installer and an installation window will appear.

3. Press the “Add Python X.X to your PATH” option and install the python.

This way you can set up a default path without any headache.

BASIC SYNTAX

The Python syntax defines a set of rules that are used to create a Python Program. The Python Programming Language Syntax has many similarities to Perl, C, and Java Programming Languages. However, there are some definite differences between the languages.

Modes of Python Programming

There are two different **modes of Python Programming**...

1. Interactive Mode Programming

2. Script Mode Programming.

Let's execute a Python program to print "Hello, World!" in both modes...

1. Interactive Mode Programming

```
>>>
```

Here >>> denotes a Python Command Prompt where you can type your commands. Let's type the following text at the Python prompt and press the Enter...

```
>>> print ("Hello, World!")
```

Result-

```
Hello, World!
```

2. Script Mode Programming

We can write a simple Python program in a script which is simple text file. Python files have extension .py. Type the following source code in a test.py file...

```
print ("Hello, World!")
```

We assume that you have Python interpreter path set in PATH variable. Now, let's try to run this program as follows...

```
test.py
```

Result-

```
Hello, World!
```

PYTHON COMMENTS

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them.

Example:

```
#This is a comment
print("Hello, World!")
```

Single line Comment

Comments can be placed at the end of a line, and Python will ignore the rest of the line.

Example:

```
print("Hello, World!") #This is a comment
```

A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code.

Example:

```
#print("Hello, World!")
print("Cheers, Mate!")
```

Multiline Comments

Python does not really have a syntax for multiline comments.

To add a multiline comment you could insert a # for each line.

Example:

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Or, not quite as intended, you can use a multiline string.

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it.

Example:

```
"""
```

This is a comment

written in

more than just one line

"""

print("Hello, World!")

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multi-line comment.

VARIABLE

The word variable suggests that it is something that varies or changes.

A variable in Python is a container or storage box of which its contents can change depending on what we put into the container or storage box.

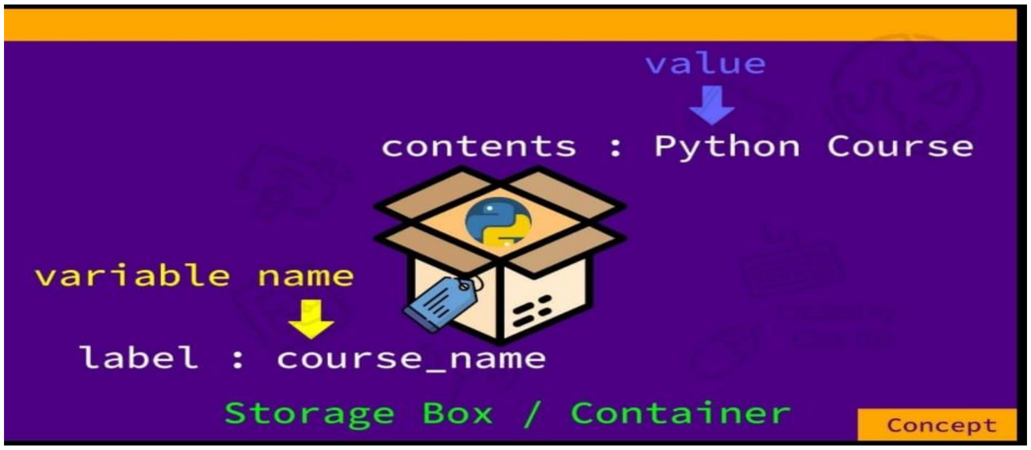
It is a storage holder that has a name or label pasted outside it and it holds something of value inside it.

We can imagine a storage box with the label course_name and inside this storage box it contains the "Python Course".

The container's label should match whatever is contained inside the container.

The container's label should clearly and appropriately describe what will be stored inside the container.

80% of programming codes consist of containers, storage boxes, or variables because computers are so good at storing things that change.



Creating Variables

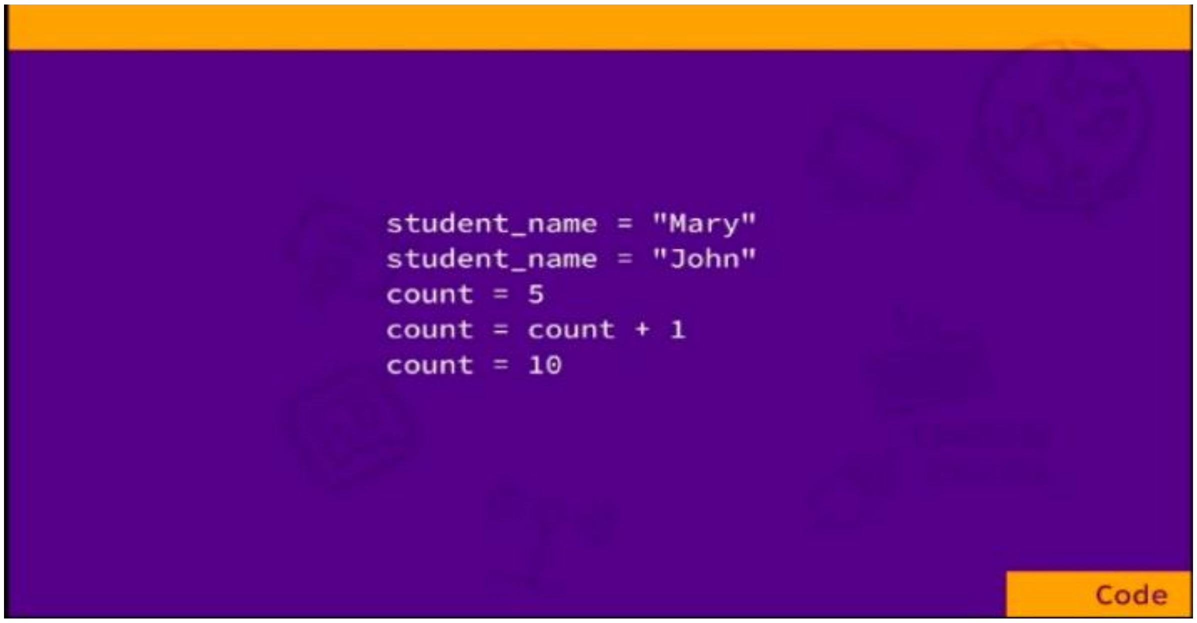
When we create a variable, we give it a name it is like when we use a storage box, the first thing we do is to label it with a name.

We use a variable name to reference data such that we can access the data inside the variable by referencing the variable's name.

The variable's name is fixed but its content or value can change it is like now we put something into the storage box and then further on we can take it out and put something else in.

Assigning Variables

To assign a value to a variable name, we use the assignment operator.



```
student_name = "Mary"
student_name = "John"
count = 5
count = count + 1
count = 10
```

The = sign is called the assignment operator.

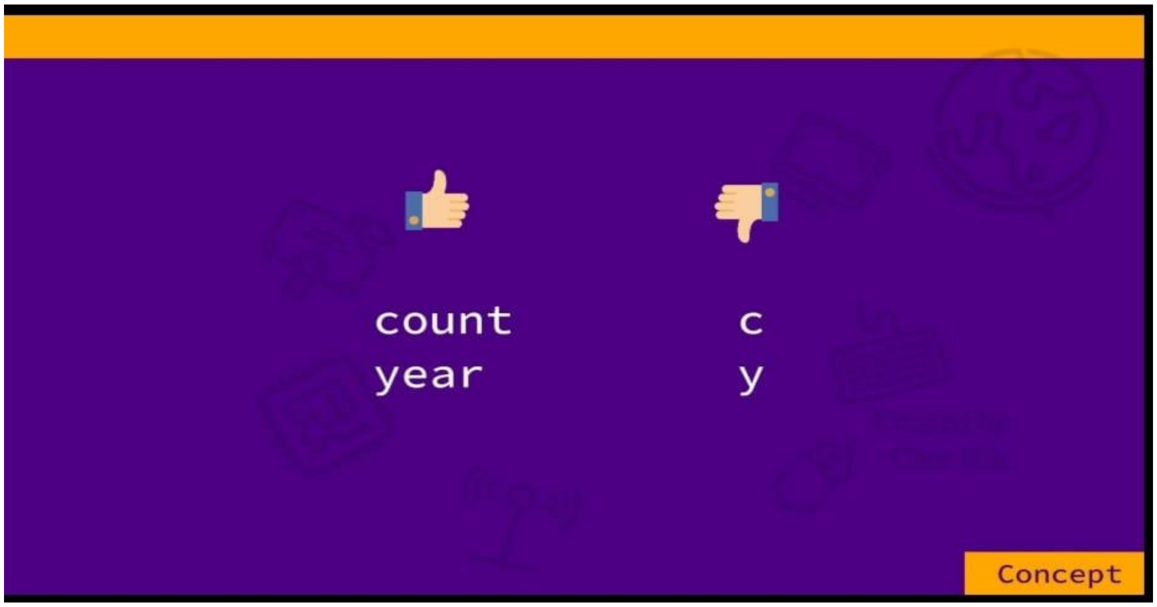
The variable name student_name has its value changed from "Mary" to "John".

The variable name count has its value changed from 5 to 6 by adding 1 to it and then changed again from 6 to 10 by taking out the 6 and putting in the 10.

Naming Variables

Select meaningful variable names that represent what they are for.

Try to use the variable name count instead of just c, and year instead of just y.

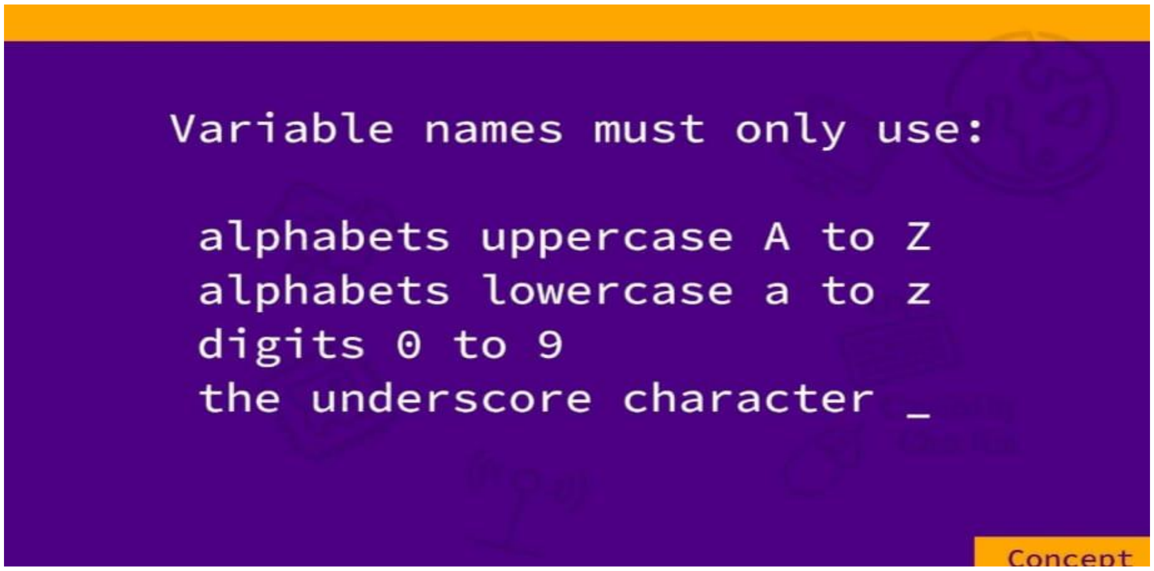


count
year

c
y

Concept

Variable name must only use:



Variable names must only use:

alphabets uppercase A to Z
alphabets lowercase a to z
digits 0 to 9
the underscore character _

Concept

Examples of valid variable names are:

student_name

_student_name

Student_name

studentname

studentName

studentName1

STUDENTNAME_

Examples of invalid variable names are:

1studentName

student@name

student name

student-name

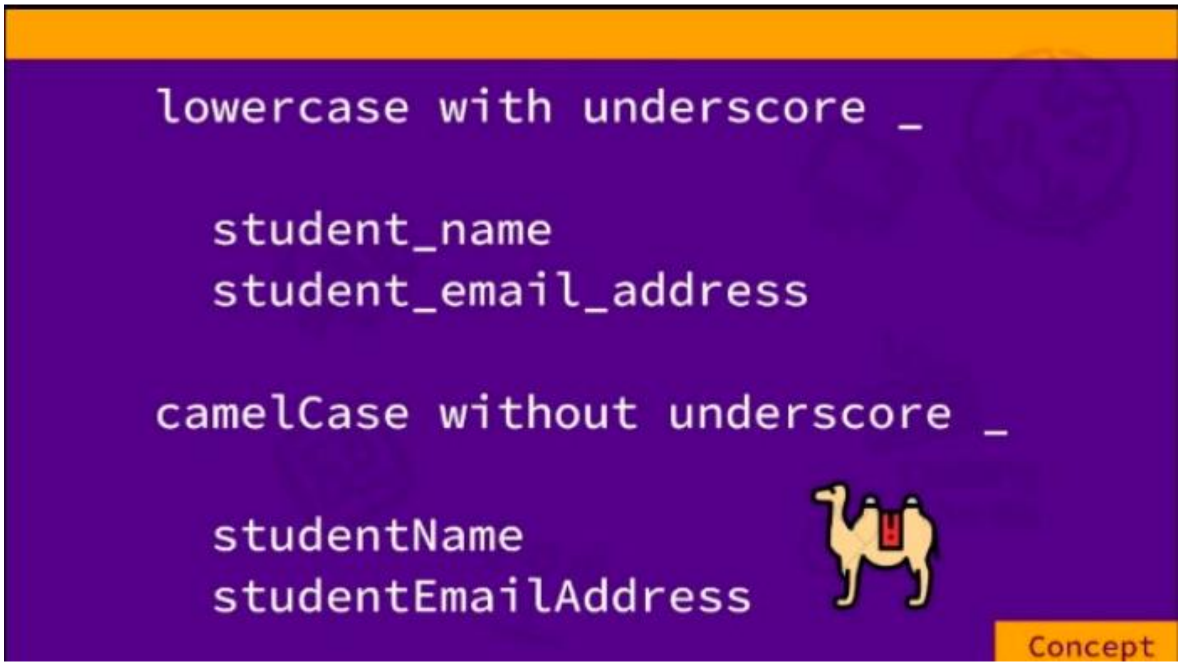
1studentName is invalid because variable names cannot start with a digit.

student@name is invalid because variable names cannot have any special characters except the underscore character

student name is invalid because variable names cannot have any space characters.

student-name is invalid because hyphens are not allowed.

The general naming convention in Python when it comes to variable names, is to use lowercase alphabets and separate each word using the underscore character_.



```
lowercase with underscore _

   student_name
   student_email_address

camelCase without underscore _

   studentName
   studentEmailAddress
```

The camelCase

However, many Python software developers use the camelCase instead of the lowercase separated by the underscore character.

camelCase refers to variable names starting with a lowercase character, and instead of using the underscore character _ to separate 2 words, it starts the next word with an uppercase character.

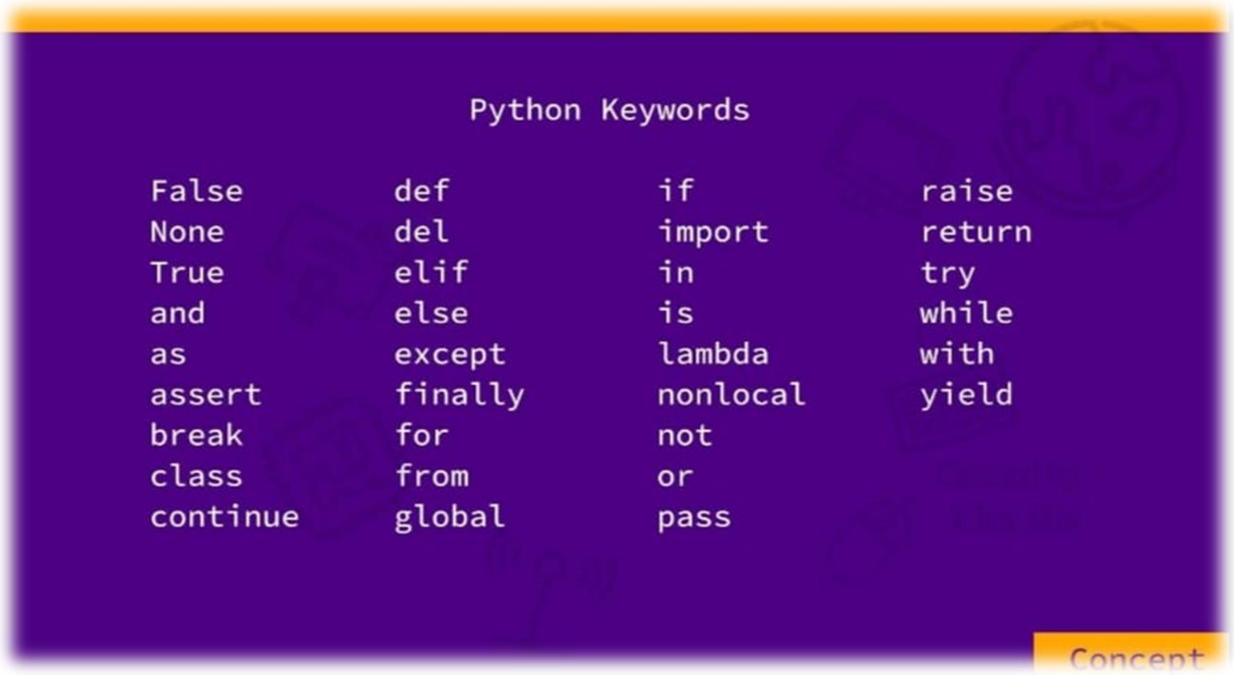
It is important to note that regardless of whether we use lowercase with underscore character _ or camelCase, we should be consistent and adopt the same naming convention throughout our program.

And variable names are case- sensitive:

student_name is not the same as STUDENT_NAME

Python Keywords

All Python keywords cannot be used for variable names:


Python Keywords

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

Python is a case-sensitive language and Python keywords are case- sensitive.

Please take note that these 3 keywords - False, None, True all of them start with an uppercase character and not a lowercase character:

False

None

True

Variable names are also case-sensitive:

False is not the same as false

False cannot be used as a variable name

false can be used as a variable name

The Python keyword False, which starts with an uppercase F, is not the same as lowercase false.

Python keyword False cannot be used as a variable name, but lowercase false can be used as a variable name.

Tracing Variables

As the value of a variable changes throughout our program because new values are constantly assigned to it, we can use the print() function to trace the value of our variables.

The print() function can be used for debugging purposes whereby we need to know what went wrong in our program - and one way to find out is to trace how the value of our variables changes in the course of our program.

DATA TYPES OF PYTHON

Data types are the classification or categorization of data items. Python supports the following built-in data types.

Scalar Types

1. int: Positive or negative whole numbers (without a fractional part) e.g. -10, 10, 456, 4654654.

2. float: Any real number with a floating-point representation in which a fractional component is denoted by a decimal symbol or scientific notation e.g. 1.23, 3.4556789e2.

3. complex: A number with a real and imaginary component represented as x + 2y.

4. bool: Data with one of two built-in values True or False. Notice that 'T' and 'F' are capital. true and false are not valid booleans and Python will throw an error for them.

None: The None represents the null object in Python. A None is returned by functions that don't explicitly return a value.

Sequence Type

A sequence is an ordered collection of similar or different data types. Python has the following built-in sequence data types...

1. String: A string value is a collection of one or more characters put in single, double or triple quotes.

2. List: A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.

3. Tuple: A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

Mapping Type

1. Dictionary: A dictionary Dict() object is an unordered collection of data in a key:value pair form. A collection of such pairs is enclosed in curly brackets. For example: {1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}

Set Types

1. set: Set is mutable, unordered collection of distinct hashable objects. The set is a Python implementation of the set in Mathematics. A set object has suitable methods to perform mathematical set operations like union, intersection, difference, etc.

2. frozenset: Frozenset is immutable version of set whose elements are added from other iterables.

Mutable and Immutable Types

Data objects of the above types are stored in a computer's memory for processing. Some of these values can be modified during processing, but contents of others can't be altered once they are created in the memory.

Numbers, strings, and Tuples are immutable, which means their contents can't be altered after creation.

On the other hand, items in a List or Dictionary object can be modified. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are mutable objects.

CASTING

Casting is a process of converting a variable's data type into another data type. If you want to specify the data type of a variable, this can be done with casting.

Example:

```
x = str(3)    # x will be '3'

y = int(3)    # y will be 3

z = float(3)  # z will be 3.0
```

PYTHON STRING

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Creating a string in Python

Syntax:

str = "Hi Python !"

Here, if we check the type of the variable str using a Python script

print(type(str)), then it will print a string (str).

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

#Using single quotes

str1 = 'Hello Python'

print(str1)

#Using double quotes

str2 = "Hello Python"

print(str2)

#Using triple quotes

str3 = '''Triple quotes are generally used for represent the multiline or docstring'''

print(str3)

Output:

Hello Python

Hello Python

Triple quotes are generally used for represent the multiline or docstring.

BOOLEAN

Booleans represent one of two values: True or False.

In programming you often need to know if an expression is True or False.

You can evaluate any expression in Python, and get one of two answers, True or False.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(10 > 9)

print(10 == 9)

print(10 < 9)
```

When you run a condition in an if statement, Python returns True or False:

Example:

Print a message based on whether the condition is True or False:

```
a = 200

b = 33

if b > a:

    print("b is greater than a")

else:

    print("b is not greater than a")
```

Evaluate Values and Variables

The bool() function allows you to evaluate any value, and give you True or False in return,

Example:

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```

Example:

Evaluate two variables:

```
x = "Hello"

y = 15

print(bool(x))

print(bool(y))
```

ALSO READ:

PYTHON OPERATORS

Operators are special symbols that perform operations on variables and values.

Example:

```
print(5 + 6)   # 11
```

Here, + is an operator that adds two numbers: 5 and 6.

Types of Python Operators

There are 7 types of Python operators...1. Arithmetic Operators

2. Assignment Operators

3. Comparison Operators

4. Logical Operators

5. Bitwise Operators

6. Identity Operators

7. Membership Operators

1. Python Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Example:

```
sub = 10 - 5 # 5
```

Here, - is an arithmetic operator that subtracts two values or variables.

2. Python Assignment Operators

Assignment operators are used to assign values to variables. For example,

```
# assign 5 to x

var x = 5
```

Here, = is an assignment operator that assigns 5 to x.

3. Python Comparison Operators

Comparison operators compare two values/variables and return a boolean result: True or False.

Example:

```
a = 5

b =2

print (a > b)    # True
```

4. Python Logical Operators

Logical operators are used to check whether an expression is True or False. They are used in decision-making. For example,

```
a = 5

b = 6

print((a > 2) and (b >= 6))    # True
```

5. Python Bitwise Operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

Example:

2 is 10 in binary, and 7 is 111.

6. Identity Operators

Identity operators are used for locating the memory unit of the objects, especially when both objects have the same name and can be differentiated only using their memory location.

Types of Identity Operators

There are two types of identity operators...

1. Is Operator

2. Is Not Operator

Is Operator

Is operator is used for comparing if the objects are in the same location while returning ‘true’ or ‘false’ values as a result.

Example:

```
m = 70

n = 70

if ( m is n ):

   print("Result: m and n have same identity")

else:

   print("Result: m and n do not have same identity")
```

Output: m and n have same identity

Is Not Operator

Is Not operator works in the opposite way of is operator? This returns true if the memory location of two objects is not the same. This returns False if the memory location of two objects is the same.

Example:

```
m = 70

n = 70

if ( m is not n ):

   print("Result: m and n have same identity")

else:

   print("Result: m and n do not have same identity")
```

Output: m and n do not have same identity

7. Membership Operators

We use membership operators to check whether a value or variable exists in a sequence (string, list, tuples, sets, dictionary) or not. Python has two membership operators: in and not in. Both return a Boolean result. The result of in operator is opposite to that of not in operator.

Example:

```
var = "TotalSach"

a = "s"

b = "tal"

c = "ac"

d = "Al"

print (a, "in", var, ":", a in var)

print (b, "not in", var, ":", b not in var)

print (c, "in", var, ":", c in var)

print (d, "not in", var, ":", d not in var)
```

Output:

```
s in TotalSach : True

tal not in TotalSach : False

ac in TotalSach : True

Al not in TotalSach : True
```

PYTHON CONDITIONAL STATEMENTS

There are situations in real life when we need to make some decisions and based on these decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.
Conditional statements in Python languages decide the direction(Control Flow) of the flow of program execution.

Types of Statements in Python

There are 4 types of Statements in Python...

1. The if statement

2. The if-else statement

3. The nested-if statement

4. The if-elif-else ladder

if statement

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

Syntax:

```
if condition:
   # Statements to execute if
   # condition is true
```

Here, the condition after evaluation will be either true or false. if the statement accepts boolean values-if the value is true then it will execute the block of statements below it otherwise not.
As we know, python uses indentation to identify a block. So the block under an if statement will be identified as shown in the below example:

```
if condition:
   statement1
statement2
# Here if the condition is true, if block
# will consider only statement1 to be inside
# its block.
```

Example:
As the condition present in the if statement is false. So, the block below the if statement is executed.

```
# python program to illustrate If else statement
#!/usr/bin/python
```

```
i = 20
if (i < 15):
    print("i is smaller than 15")
    print("i'm in if Block")
else:
    print("i is greater than 15")
    print("i'm in else Block")
print("i'm not in if and not in else Block")
```

Output:

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

if-else Statement

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But if we want to do something else if the condition is false, we can use the else statement with the if statement to execute a block of code when the if condition is false.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

Example:
The block of code following the else statement is executed as the condition present in the if statement is false after calling the statement which is not in the block(without spaces).

```
# Explicit function
def digitSum(n):
    dsum = 0
    for ele in str(n):
        dsum += int(ele)
    return dsum

# Initializing list
List = [367, 111, 562, 945, 6726, 873]

# Using the function on odd elements of the list
newList = [digitSum(i) for i in List if i & 1]
```

```
# Displaying new list
print(newList)
```

Output :

```
[16, 3, 18, 18]
```

Nested-if Statement

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement.
Yes, Python allows us to nest if statements within if statements. i.e., we can place an if statement inside another if statement.

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Example:
In this example, we are showing nested if conditions in the code, All the If conditions will be executed one by one.

```
# python program to illustrate nested If statement

i = 10
if (i == 10):

    #  First if statement
    if (i < 15):
        print("i is smaller than 15")

    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if (i < 12):
        print("i is smaller than 12 too")
    else:
        print("i is greater than 15")
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

if-elif-else Ladder

Here, a user can decide among multiple options. The if statements are executed from the top down.
As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final “else” statement will be executed.

Syntax:

```
if (condition):
    statement
elif (condition):
    statement
else:
    statement
```

Example:
In the example, we are showing single if condition, multiple elif conditions, and single else condition.

```
# Python program to illustrate if-elif-else ladder
#!/usr/bin/python

i = 20
if (i == 10):
    print("i is 10")
elif (i == 15):
    print("i is 15")
elif (i == 20):
    print("i is 20")
else:
    print("i is not present")
```

Output:

```
i is 20
```

LOOPING

The following loops are available in Python to fulfil the looping needs. Python offers 3 choices for running the loops. The basic functionality of all the techniques is the same, although the syntax and the amount of time required for checking the condition differ.

We can run a single statement or set of statements repeatedly using a loop command.

The following sorts of loops are available in the Python programming language.

Sr. No.	Name of the loop	Loop Type & Description
1	**While loop**	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**For loop**	This type of loop executes a code block multiple times and abbreviates the code that manages the loop variable.
3	**Nested loops**	We can iterate a loop inside another loop.

Loop Control Statements

Statements used to control loops and change the course of iteration are called control statements. All the objects produced within the local scope of the loop are deleted when execution is completed.

Python provides the following control statements. We will discuss them later in detail.

Let us quickly go over the definitions of these loop control statements.

Sr.No.	Name of the control statement	Description
1	**Break statement**	This command terminates the loop's execution and transfers the program's control to the statement next to the loop.
2	**Continue**	This command skips the current iteration of the loop. The statements following

	statement	the continue statement are not executed once the Python interpreter reaches the continue statement.
3	**Pass statement**	The pass statement is used when a statement is syntactically necessary, but no code is to be executed.

The for Loop

Python's for loop is designed to repeatedly execute a code block while iterating through a list, tuple, dictionary, or other iterable objects of Python. The process of traversing a sequence is known as iteration.

Syntax of the for Loop

```
for value in sequence:
    { code block }
```

In this case, the variable value is used to hold the value of every item present in the sequence before the iteration begins until this particular iteration is completed.

Loop iterates until the final item of the sequence are reached.

Code

```
# Python program to show how the for loop works

# Creating a sequence which is a tuple of numbers
numbers = [4, 2, 6, 7, 3, 5, 8, 10, 6, 1, 9, 2]

# variable to store the square of the number
square = 0

# Creating an empty list
squares = []

# Creating a for loop
for value in numbers:
    square = value ** 2
    squares.append(square)
```

16. **print**("The list of squares is", squares)

Output:

```
The list of squares is [16, 4, 36, 49, 9, 25, 64, 100, 36, 1,
 81, 4]
```

PYTHON LOOP CONTROL STATEMENTS

Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Python Continue
It returns the control to the beginning of the loop.

- Python3

```
# Prints all letters except 'e' and 's'

for letter in 'geeksforgeeks':

    if letter == 'e' or letter == 's':

        continue

    print('Current Letter :', letter)
```

Output:

```
Current Letter : g

Current Letter : k

Current Letter : f

Current Letter : o

Current Letter : r

Current Letter : g

Current Letter : k
```

Python Break
It brings control out of the loop.

- Python3

```
for letter in 'geeksforgeeks':



    # break the loop as soon it sees 'e'

    # or 's'

    if letter == 'e' or letter == 's':

        break

print('Current Letter :', letter)
```

Output:

Current Letter : e

Python Pass
We use pass statements to write empty loops. Pass is also used for empty control statements, functions, and classes.

- Python3

```
# An empty loop

for letter in 'geeksforgeeks':

    pass

print('Last Letter :', letter)
```

Output:
Last Letter :

PYTHON STRING

Till now, we have discussed numbers as the standard data-types in Python. In this section of the tutorial, we will discuss the most popular data type in Python, i.e., string.

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.

Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Consider the following example in Python to create a string.

Syntax:

```
str = "Hi Python !"
```

Here, if we check the type of the variable **str** using a Python script

```
print(type(str)), then it will print a string (str).
```

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

Creating String in Python

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or **docstrings**.

```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
print(str2)

#Using triple quotes
```

```
str3 = """Triple quotes are generally used for
    represent the multiline or
    docstring'''
print(str3)
```

Output:

```
Hello Python
Hello Python
Triple quotes are generally used for
    represent the multiline or
    docstring
```

PYTHON LIST

In Python, the sequence of various data types is stored in a list. A list is a collection of different kinds of values or items. Since Python lists are mutable, we can change their elements after forming. The comma (,) and the square brackets [enclose the List's items] serve as separators.

Although six Python data types can hold sequences, the List is the most common and reliable form. A list, a type of sequence data, is used to store the collection of data. Tuples and Strings are two similar data formats for sequences.

Lists written in Python are identical to dynamically scaled arrays defined in other languages, such as Array List in Java and Vector in C++. A list is a collection of items separated by commas and denoted by the symbol [].

List Declaration

Code

```
# a simple list
list1 = [1, 2, "Python", "Program", 15.9]
list2 = ["Amy", "Ryan", "Henry", "Emma"]

# printing the list
print(list1)
print(list2)

# printing the type of list
print(type(list1))
print(type(list2))
```

Output:

```
[1, 2, 'Python', 'Program', 15.9]
['Amy', 'Ryan', 'Henry', 'Emma']
< class ' list ' >
< class ' list ' >
```

Characteristics of Lists

The characteristics of the List are as follows:

- The lists are in order.
- The list element can be accessed via the index.
- The mutable type of List is
- The rundowns are changeable sorts.
- The number of various elements can be stored in a list.

Ordered List Checking

Code

```
# example
a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6 ]
b = [ 1, 2, 5, "Ram", 3.50, "Rahul", 6 ]
a == b
```

Output:

```
False
```

Python Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In simple language, a list is a collection of things, enclosed in [ ] and separated by commas.
The list is a sequence data type which is used to store the collection of data. Tuples and String are other types of sequence data types.

Example of list in Python
Here we are creating Python **List** using [].

- Python3

```
Var = ["Geeks", "for", "Geeks"]

print(Var)
```

Output:

```
["Geeks", "for", "Geeks"]
```

Python Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In simple language, a list is a collection of things, enclosed in [ ] and separated by commas.
The list is a sequence data type which is used to store the collection of data. Tuples and String are other types of sequence data types.

Python Lists

```
mylist = ["apple", "banana", "cherry"]
```

List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

OUTPUT

```
['apple', 'banana', 'cherry']
```

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Note: There are some list methods that will change the order, but in general: the order of the items will not change.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

OUTPUT

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

PYTHON TUPLES

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

*Example*Get your own Python Server

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Python Sets

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

*** Note:** Set *items* are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Example

Create a Set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

```
{'banana', 'cherry', 'apple'}
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Once a set is created, you cannot change its items, but you can remove items and add new items.

Duplicates Not Allowed

Sets cannot have two items with the same value.

PYTHON DICTIONARIES

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

OUTPUT

```
{'brand': 'Ford', 'model': 'Mustang', 'year'
```

Dictionary Items

Dictionary items are ordered, changeable, and do not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"]
```

OUTPUT

```
Ford
```

ARRAYS

Note: This page shows you how to use LISTS as ARRAYS, however, to work with arrays in Python you will have to import a library, like the NumPy library.

Arrays are used to store multiple values in one single variable:

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

OUTPUT

```
['Ford', 'Volvo', 'BMW']
```

What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
x = cars[0]
```

Example

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

OUTPUT

VE

The Length of an Array

Use the len() method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the cars array:

```
x = len(cars)
```

Looping Array Elements

You can use the for in loop to loop through all the elements of an array.

Example

Print each item in the cars array:

```
for x in cars:
  print(x)
```

Adding Array Elements

You can use the append() method to add an element to an array.

Example

Add one more element to the cars array:

```
cars.append("Honda")
```

Removing Array Elements

You can use the pop() method to remove an element from the array.

Example

Delete the second element of the cars array:

```
cars.pop(1)
```

PYTHON ITERATORS

Python Iterators

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.

All these objects have a iter() method which is used to get an iterator:

ExampleGet your own Python Server

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

OUTPUT

```
APPLE

BABANA

CHERRY
```

Even strings are iterable objects, and can return an iterator:

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

OUTPUT

```
B

A

N

A

N

A
```

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Create an Iterator

To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called __init__(), which allows you to do some initializing when the object is being created.

The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The __next__() method also allows you to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

PYTHON MODULES

Create a Module

To create a module just save the code you want in a file with the file extension .py:

Python Modules

In this tutorial, we will explain how to construct and import custom Python modules. Additionally, we may import or integrate Python's built-in modules via various methods.

What is Modular Programming?

Modular programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. We call these subtasks modules. Therefore, we can build a bigger program by assembling different modules that act like building blocks.

Modularizing our code in a big application has a lot of benefits.

Simplification: A module often concentrates on one comparatively small area of the overall problem instead of the full task. We will have a more manageable design problem to think about if we are only concentrating on one module. Program development is now simpler and much less vulnerable to mistakes.

Flexibility: Modules are frequently used to establish conceptual separations between various problem areas. It is less likely that changes to one module would influence other portions of the program if modules are constructed in a fashion that reduces interconnectedness. (We might even be capable of editing a module despite being familiar with the program beyond it.) It increases the likelihood that a group of numerous developers will be able to collaborate on a big project.

Reusability: Functions created in a particular module may be readily accessed by different sections of the assignment (through a suitably established api). As a result, duplicate code is no longer necessary.

Scope: Modules often declare a distinct namespace to prevent identifier clashes in various parts of a program.

In Python, modularization of the code is encouraged through the use of functions, modules, and packages.

What are Modules in Python?

A document with definitions of functions and various statements written in Python is called a Python module.

In Python, we can define a module in one of 3 ways:

- Python itself allows for the creation of modules.
- Similar to the re (regular expression) module, a module can be primarily written in C programming language and then dynamically inserted at run-time.
- A built-in module, such as the itertools module, is inherently included in the interpreter.

A module is a file containing Python code, definitions of functions, statements, or classes. An example_module.py file is a module we will create and whose name is example_module.

We employ modules to divide complicated programs into smaller, more understandable pieces. Modules also allow for the reuse of code.

Rather than duplicating their definitions into several applications, we may define our most frequently used functions in a separate module and then import the complete module.

Let's construct a module. Save the file as example_module.py after entering the following.

Example:

```python
# Here, we are creating a simple Python program to show how to create a module.
# defining a function in the module to reuse it
def square( number ):
  # here, the above function will square the number passed as the input
   result = number ** 2
   return result     # here, we are returning the result of the function
```

Here, a module called example_module contains the definition of the function square(). The function returns the square of a given number.

How to Import Modules in Python?

In Python, we may import functions from one module into our program, or as we say into, another module.

For this, we make use of the import Python keyword. In the Python window, we add the next to import keyword, the name of the module we need to import. We will import the module we defined earlier example_module.

Syntax:

```
import example_module
```

The functions that we defined in the example_module are not immediately imported into the present program. Only the name of the module, i.e., example_ module, is imported here.

We may use the dot operator to use the functions using the module name. For instance:

Example:

```
# here, we are calling the module square method and passing the value 4
result = example_module.square(  4  )
print("By using the module square of number is: ", result )
```

Output:

```
By using the module square of number is: 16
```

There are several standard modules for Python. The complete list of Python standard modules is available. The list can be seen using the help command.

Similar to how we imported our module, a user-defined module, we can use an import statement to import other standard modules.

Importing a module can be done in a variety of ways. Below is a list of them.

Python import Statement

Using the import Python keyword and the dot operator, we may import a standard module and can access the defined functions within it. Here's an illustration.

Code

```
# Here, we are creating a simple Python program to show how to import a standard module
# Here, we are import the math module which is a standard module
import math
```

```
4. print( "The value of euler's number is", math.e )
5. # here, we are printing the euler's number from the math module
```

Output:

```
The value of euler's number is 2.718281828459045
```

PYTHON DATE

Python provides the **datetime** module work with real dates and times. In real-world applications, we need to work with the date and time. Python enables us to schedule our Python script to run at a particular timing.

In Python, the date is not a data type, but we can work with the date objects by importing the module named with **datetime, time, and calendar**.

In this section of the tutorial, we will discuss how to work with the date and time objects in Python.

The **datetime** classes are classified in the six main classes.

- **date** - It is a naive ideal date. It consists of the year, month, and day as attributes.
- **time** - It is a perfect time, assuming every day has precisely 24*60*60 seconds. It has hour, minute, second, microsecond, and **tzinfo** as attributes.
- **datetime** - It is a grouping of date and time, along with the attributes year, month, day, hour, minute, second, microsecond, and tzinfo.
- **timedelta -** It represents the difference between two dates, time or datetime instances to microsecond resolution.
- **tzinfo** - It provides time zone information objects.
- **timezone -** It is included in the new version of Python. It is the class that implements the **tzinfo** abstract base class.

❖ Tick

In Python, the time instants are counted since 12 AM, 1st January 1970. The function **time()** of the module time returns the total number of ticks spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

Consider the following example

```
import time;
#prints the number of ticks spent since 12 AM, 1st January 1970
print(time.time())
```

Output:

```
1585928913.6519969
```

How to get the current time?

The localtime() functions of the time module are used to get the current time tuple. Consider the following exam.

PYTHON MATH

Python Math Module

Introduction.**In this article, we are discussing Math Module in Python. We can easily calculate many mathematical calculations in Python using the Math module. Mathematical calculations may occasionally be required when dealing with specific fiscal or rigorous scientific tasks. Python has a math module that can handle these complex calculations. The functions in the math module can perform simple mathematical calculations like addition (+) and subtraction (-) and advanced mathematical calculations like trigonometric operations and logarithmic operations.**

This tutorial teaches us about applying the math module from fundamentals to more advanced concepts with the support of easy examples to understand the concepts fully. We have included the list of all built-in functions defined in this module for better understanding.

What is Math Module in Python?

Python has a built-in math module. It is a standard module, so we do not need to install it separately. We only must import it into the program we want to use. We can import the module, like any other module of Python, using import math to implement the functions to perform mathematical operations.

Since the source code of this module is in the C language, it provides access to the functionalities of the underlying C library. Here we have given some basic examples of the Math module in Python. The examples are written below -

Program Code 1:

Here we give an example of a math module in Python for calculating the square root of a number. The code is given below -

```
# This program will show the calculation of square root using the math module
# importing the math module
import math
print(math.sqrt( 9 ))
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
3.0
```

This Python module does not accept complex data types. The more complicated equivalent is the math module.

We can, for example, calculate all trigonometric ratios for any given angle using the built-in functions in the math module. We must provide angles in radians to these trigonometric functions (sin, cos, tan, etc.). However, we are accustomed to measuring angles in terms of degrees. The math module provides two methods to convert angles from radians to degrees and vice versa.

Program code 2:

Here we give an example of a math module in Python for calculating the factorial of a number. The code is given below -

```
import math
n=int(input())
print(math.factorial(n))
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
5
120
```

Python User Input

User Input

Python allows for user input.

That means we are able to ask the user for input.

The method is a bit different in Python 3.6 than Python 2.7.

Python 3.6 uses the input() method.

Python 2.7 uses the raw_input() method.

The following example asks for the username, and when you entered the username, it gets printed on the screen:

Python 3.6

```
username = input("Enter username:")
print("Username is: " + username)
```

OUTPUT

INTER USERNAME:

Python input

How to Take Input from User in Python
Sometimes a developer might want to take user input at some point in the program. To do this Python provides an input() function.

Syntax:
input('prompt')

where prompt is an optional string that is displayed on the string at the time of taking input.

Example 1: Python get **user input with a message**

- Python3

```
# Taking input from the user

name = input("Enter your name: ")
```

```
# Output

print("Hello, " + name)

print(type(name))
```

Output:

```
Enter your name: GFG

Hello, GFG

<class 'str'>
```

Note: Python takes all the input as a string input by default. To convert it to any other data type we have to convert the input explicitly. For example, to convert the input to int or float we have to use the int() and float() method respectively.

Example 2: **Integer input in Python**

- Python3

```
# Taking input from the user as integer

num = int(input("Enter a number: "))



add = num + 1



# Output

print(add)
```

Output:

```
Enter a number: 25

26
```

How to take Multiple Inputs in Python :

we can take multiple inputs of the same data type at a time in python, using map() method in python.

- Python3

```
a, b, c = map(int, input("Enter the Numbers : ").split())

print("The Numbers are : ",end = " ")

print(a, b, c)
```

Output :

```
Enter the Numbers : 2 3 4

The Numbers are :  2 3 4
```

Python output

How to Display Output in Python

Python provides the print() function to display output to the standard output devices.

Syntax: print(value(s), sep= ‘ ‘, end = ‘\n’, file=file, flush=flush)

Parameters:

value(s) : Any value, and as many as you like. Will be converted to string before printed
sep=’separator’ : (Optional) Specify how to separate the objects, if there is more than one.Default :’ ‘
end=’end’: (Optional) Specify what to print at the end.Default : ‘\n’
file : (Optional) An object with a write method. Default :sys.stdout
flush : (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

Returns: It returns output to the screen.

Example: Python Print Output

- Python3

```
# Python program to demonstrate
# print() method

print("GFG")



# code for disabling the softspace feature

print('G', 'F', 'G')
```

Output

```
GFG

G F G
```

In the above example, we can see that in the case of the 2nd print statement there is a space between every letter and the print statement always add a new line character at the end of the string. This is because after every character the sep parameter is printed and at the end of the string the end parameter is printed. Let’s try to change this sep and end parameter.

Example: Python Print output with custom sep and end parameter

- Python3

```
# Python program to demonstrate

# print() method

print("GFG", end = "@")



# code for disabling the softspace feature

print('G', 'F', 'G', sep="#")
```

Output
GFG@G#F#G

Formatting Output
Formatting output in Python can be done in many ways. Let’s discuss them below

Using formatted string literals

We can use **formatted string literals**, by starting a string with f or F before opening quotation marks or triple quotation marks. In this string, we can write Python expressions between **{** and **}** that can refer to a variable or any literal value.
Example: Python String formatting using F string

- Python3

```
# Declaring a variable

name = "Gfg"



# Output

print(f'Hello {name}! How are you?')
```

Output:
Hello Gfg! How are you?

FUNCTIONS & ARGUMENTS

In Python, functions are blocks of organized, reusable code that perform a specific task.. They allow you to break down your program into smaller, modular pieces, making your code more organized, easier to read, and easier to maintain. Functions can take arguments, which are inputs that the function operates on, and they can return a value as a result of their operation.

Here's a breakdown of functions and arguments in Python:

1. Defining a Function:

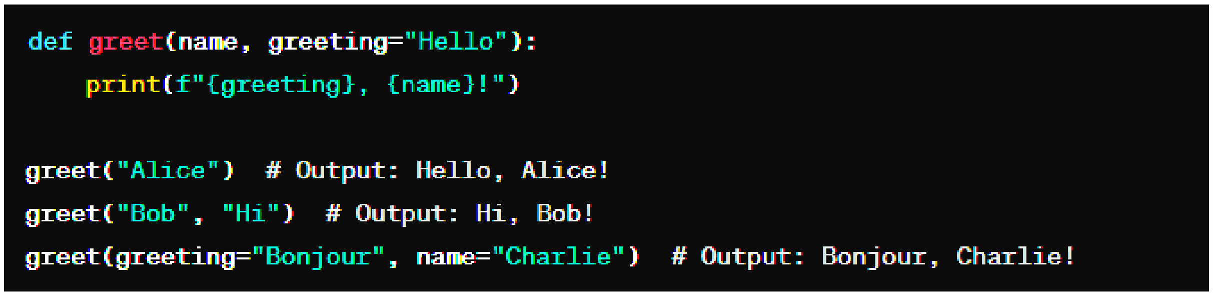
- You define a function in Python using the def keyword followed by the function name and parentheses (). If the function takes arguments, you list them inside the parentheses. The function body is then indented below the definition. Here's a basic example:

  ```
  def greet():
      print("Hello, world!")
  ```

-

2. Function Arguments:

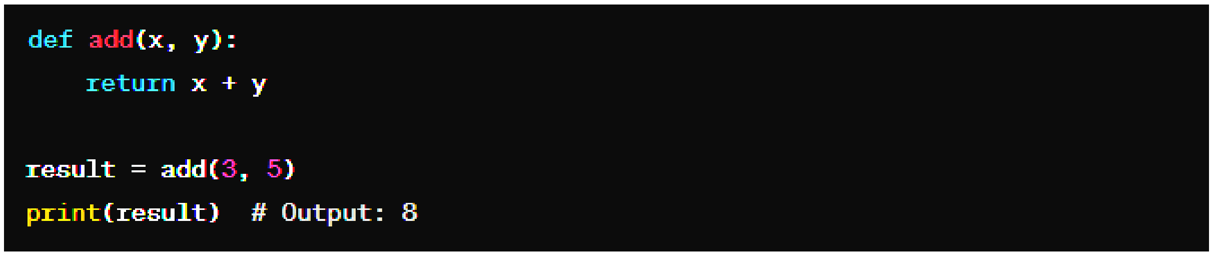
- Functions can take arguments, which are values passed into the function when it is called. These arguments provide input data to the function for it to operate on. Arguments are specified within the parentheses following the function name. There are different types of arguments in Python:
  - Positional Arguments: These are arguments that are matched based on their position in the function call. The order of the arguments matters.
  - Keyword Arguments: These are arguments preceded by a keyword when calling a function. The order of keyword arguments does not matter.
  - Default Arguments: These are arguments that have a default value specified in the function definition. If the caller doesn't provide a value for these arguments, the default value is used.
  - Variable-Length Arguments: Functions can accept a variable number of arguments. This is achieved using *args for positional arguments and **kwargs for keyword arguments.
- Here's an example illustrating these different types of arguments:

- 

```python
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")  # Output: Hello, Alice!
greet("Bob", "Hi")  # Output: Hi, Bob!
greet(greeting="Bonjour", name="Charlie")  # Output: Bonjour, Charlie!
```

3. Returning Values:

- Functions can return a value using the return keyword. This value can be of any data type, including numbers, strings, lists, dictionaries, etc. If a function doesn't explicitly return a value, it implicitly returns None.



```python
def add(x, y):
    return x + y

result = add(3, 5)
print(result)  # Output: 8
```

Functions and arguments are fundamental concepts in Python programming, enabling code reuse, modularity, and abstraction. They allow you to write efficient and maintainable code by breaking it down into smaller, manageable parts.

MODULES

In Python, a module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, you can define functions, classes, and variables, and you can also include executable code.
Modules are used to organize code into reusable units. They help in organizing Python code logically and efficiently. You can think of a module as a library or a toolbox that contains related functions and classes.

Here's a basic example of a Python module:

```python
# module.py

def greet(name):
    print("Hello, {}!".format(name))

def add(a, b):
    return a + b

pi = 3.14159
```

In this example, ‘module.py’ is a module containing a ‘greet’ function, an ‘add’ function, and a variable ‘pi.’ To use these definitions in another Python script or module, you can import them using the ‘import’ statement:

```python
# main.py

import module

module.greet("Alice")
result = module.add(2, 3)
print("Result:", result)
print("Value of pi:", module.pi)
```

.When ‘main.py’ is executed, it imports the ‘module’ module and then uses its functions and variables as needed.

Python also allows importing specific functions or variables from a module, rather than importing the entire module:

```python
# main.py

from module import greet, add, pi

greet("Bob")
result = add(4, 5)
print("Result:", result)
print("Value of pi:", pi)
```

This way, only the specified functions and variables are imported into the current namespace. Additionally, Python provides a special module called ‘__init__.py’ which can be used to initialize packages. This module can contain initialization code or be left empty. It is automatically executed when the package is imported.

```
# __init__.py

# Initialization code for the package
```

These are the basics of modules in Python. They play a crucial role in structuring and organizing Python code into reusable and manageable units.

EXCEPTION HANDLING

Exception handling in Python allows you to gracefully manage errors or exceptional situations that may occur during the execution of your code. Python provides a way to catch and handle these exceptions using ‘try’, ‘except’, ‘else’, and ‘finally’ blocks.

Here's an overview of each component:

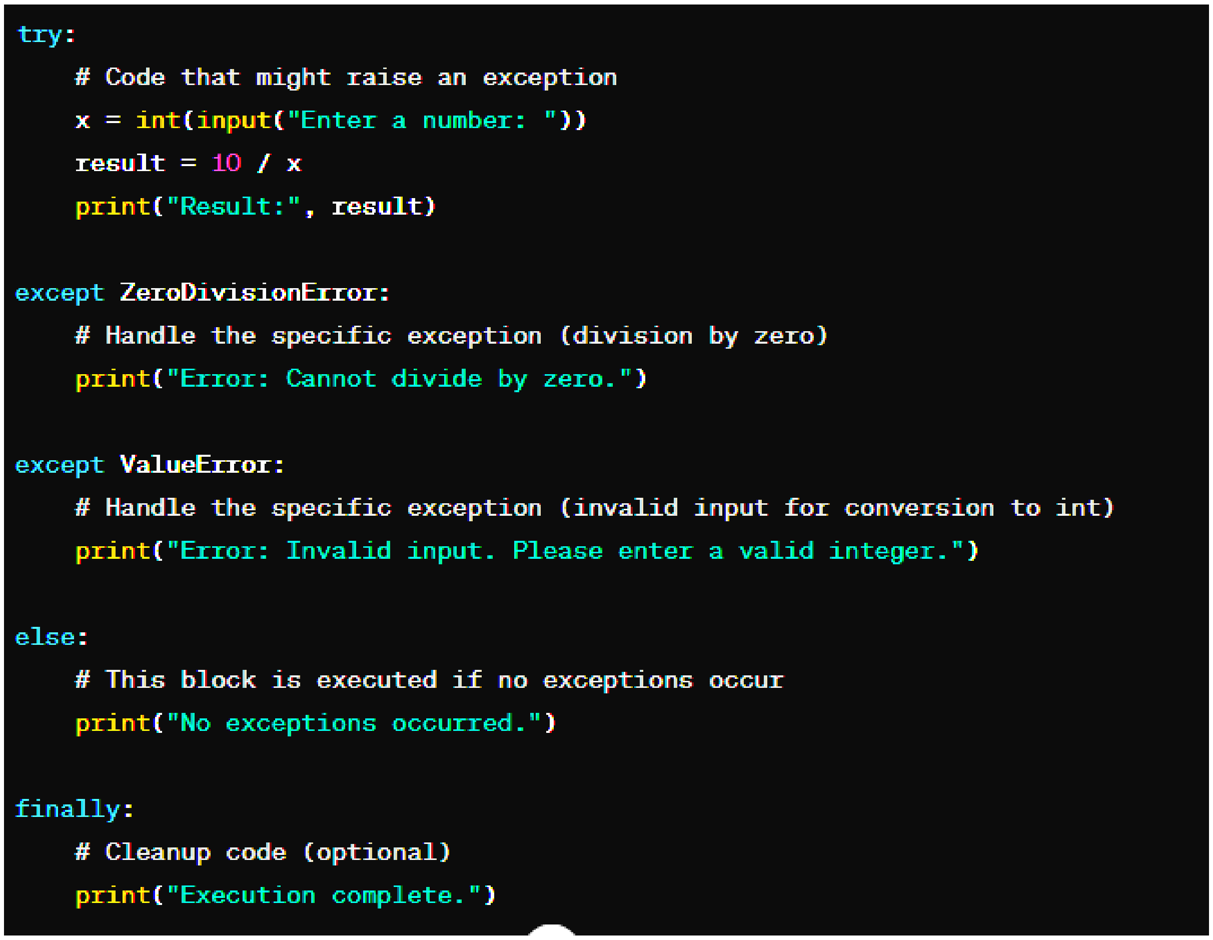
- try: The ‘try’ block is used to enclose the code that might raise an exception.

except: The ‘except’ block is used to catch and handle specific exceptions that occur within the corresponding ‘try’ block. You can specify the type of exception you want to handle, or you can use a generic ‘except’ block to catch any **exception**.

else: The ‘else’ block is executed if no exceptions occur in the ‘try’ block. It is optional.

finally: The ‘finally’ block is used to execute cleanup code that should be run regardless of whether an exception occurred. It is executed whether an exception occurred or not. It is optional.

Here's a basic example demonstrating how to use exception handling in Python:



```python
try:
    # Code that might raise an exception
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)

except ZeroDivisionError:
    # Handle the specific exception (division by zero)
    print("Error: Cannot divide by zero.")

except ValueError:
    # Handle the specific exception (invalid input for conversion to int)
    print("Error: Invalid input. Please enter a valid integer.")

else:
    # This block is executed if no exceptions occur
    print("No exceptions occurred.")

finally:
    # Cleanup code (optional)
    print("Execution complete.")
```

In this example:

The ‘try’ block contains code that prompts the user to enter a number, performs a division operation, and prints the result.

The ‘except’ blocks handle specific exceptions (‘ZeroDivisionError’ and ‘ValueError’) that may occur within the try block.

The ‘else’ block prints a message if no exceptions occur.

The ‘finally’ block prints a message indicating that the execution is complete, regardless of whether an exception occurred or not.

Exception handling allows your program to recover from errors gracefully, ensuring that it doesn't crash unexpectedly and providing feedback or alternative paths when errors occur.

<u>BUILT IN FUNCTIONS IN PYTHON</u>

Certainly! Python comes with a wide range of built-in functions that are readily available for use

without needing to import any modules. These functions serve various purposes and are foundational to programming in Python. Here are some of the most commonly used built-in functions in Python:

- 1. print(): Used to display the output to the standard output device (usually the console).
- 2. input(): Reads input from the user through the standard input device (usually the keyboard).
- 3. len(): Returns the length (the number of items) of an object. Works with sequences such as strings, lists, tuples, etc.
- 4. type(): Returns the type of an object.
- 5. int(), float(), str(), list(), tuple(), dict(), set(): These functions are used for type conversion. They convert the given value to the specified data type.
- 6. range(): Generates a sequence of numbers. It is often used with loops to iterate a certain number of times.
- 7. sum(): Returns the sum of all elements in a iterable (such as a list).
- 8. max(), min(): Returns the maximum or minimum value from a sequence or set of values.
- 9. abs(): Returns the absolute value of a number.
- 10. round(): Rounds a floating-point number to a specified number of decimal places.
- 11. sorted(): Returns a new sorted list from the elements of any iterable.
- 12. enumerate(): Returns an enumerate object that yields pairs of indexes and elements, useful for obtaining an index along with each element from an iterable.
- 13. zip(): Combines elements from multiple iterables into tuples.
- 14. map(): Applies a function to all the items in an input list or any other iterable.
- 15. filter(): Constructs an iterator from elements of an iterable for which a function returns true.

- 16. all(): Returns True if all elements of an iterable are true (or if the iterable is empty).
- 17. any(): Returns True if any element of an iterable is true. If the iterable is empty, returns False.
- 18. eval(): Evaluates a Python expression passed as a string.
- 19. format(): Formats a specified value into a specified format.
- 20. getattr(), setattr(), delattr(): Used to get, set, or delete an attribute from an object by name.

These are just a few examples of the many built-in functions available in Python. They provide a solid foundation for performing various operations and manipulations in Python programming.

FILE HANDLING IN PYTHON

File handling in Python refers to the process of working with files on a computer's filesystem. Python provides several built-in functions and modules to facilitate file handling operations. Here's an overview of the basic file handling operations in Python:

- 1. Opening a File: To open a file in Python, you use the 'open()' function, which returns a file object. The 'open()' function requires at least one argument, which is the path to the file you want to open. Optionally, you can specify the mode in which you want to open the file (read, write, append, etc.).

```
file = open('example.txt', 'r')  # Opens the file 'example.txt' in read mode
```

1. Reading from a File: Once you have opened a file, you can read its contents using various methods such as' read()', 'readline()', or 'readlines()'.

```
content = file.read()  # Reads the entire content of the file
```

```
line = file.readline()  # Reads a single line from the file
```

```
lines = file.readlines()  # Reads all lines from the file and returns a list
```

- **1. Writing to a File:** To write data to a file, you need to open the file in write mode ('w'). You can then use the 'write()' method to write data to the file.

```
file = open('example.txt', 'w')  # Opens the file 'example.txt' in write mode
file.write('Hello, World!')  # Writes the string 'Hello, World!' to the file
```

- **1. Closing a File:** After performing file operations, it's important to close the file using the 'close()' method. This releases system resources associated with the file.

```
file.close()  # Closes the file
```

- **1. Appending to a File:** To append data to an existing file, you can open the file in append mode ('a') and use the' write()' method.

```
file = open('example.txt', 'a')  # Opens the file 'example.txt' in append mode
file.write('Appending this line.')  # Appends the string to the file
file.close()  # Don't forget to close the file
```

- **1. Using** with Statement: Python's with statement is preferred for file handling as it automatically closes the file when the block is exited, ensuring proper cleanup.

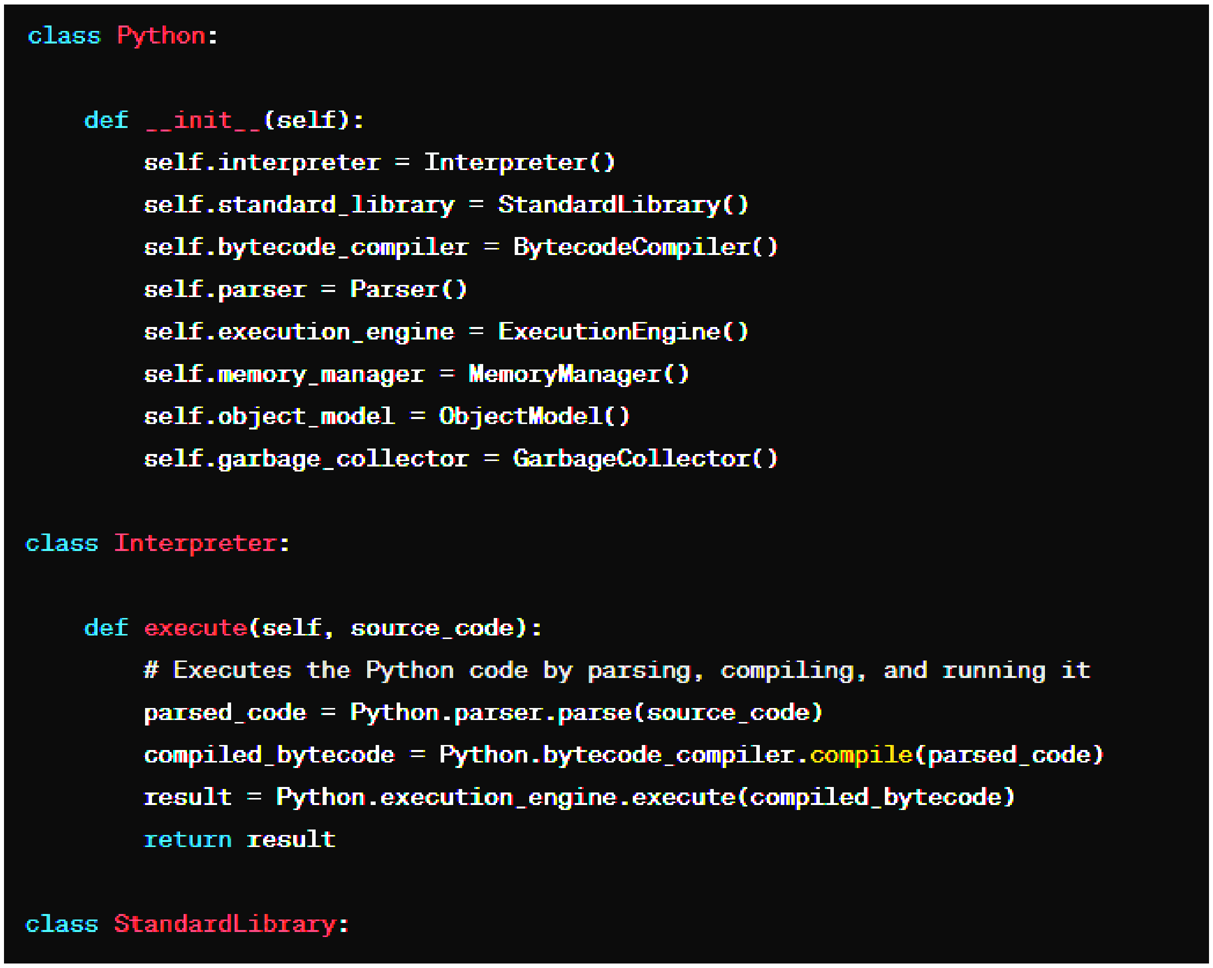
```
with open('example.txt', 'r') as file:
    content = file.read()
    # File is automatically closed when exiting the block
```

- These are the fundamental operations for handling files in Python. Additionally, Python also provides modules like 'os, shutil', and 'os.path' for performing more advanced file-related operations like file manipulation, directory handling, and path manipulations.

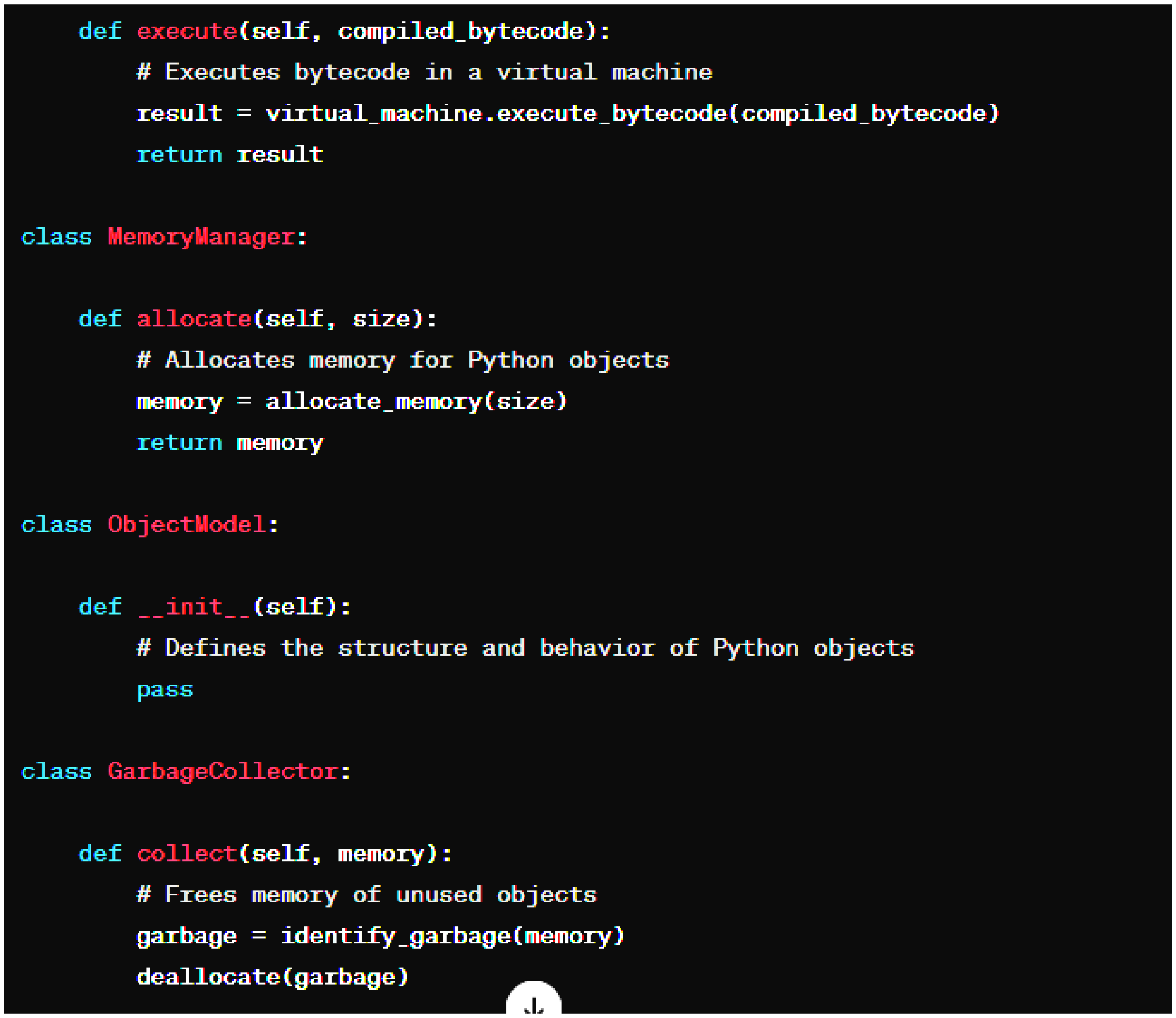
-

PYTHON ARCHITECTURE

Certainly! Below is a high-level description of Python's architecture written in Python-style pseudocode



```python
class Python:

    def __init__(self):
        self.interpreter = Interpreter()
        self.standard_library = StandardLibrary()
        self.bytecode_compiler = BytecodeCompiler()
        self.parser = Parser()
        self.execution_engine = ExecutionEngine()
        self.memory_manager = MemoryManager()
        self.object_model = ObjectModel()
        self.garbage_collector = GarbageCollector()

class Interpreter:

    def execute(self, source_code):
        # Executes the Python code by parsing, compiling, and running it
        parsed_code = Python.parser.parse(source_code)
        compiled_bytecode = Python.bytecode_compiler.compile(parsed_code)
        result = Python.execution_engine.execute(compiled_bytecode)
        return result

class StandardLibrary:
    def __init__(self):
        # Contains built-in modules and functions
        pass

class BytecodeCompiler:

    def compile(self, parsed_code):
        # Transforms parsed Python code into bytecode
        bytecode = bytecode_generation(parsed_code)
        return bytecode

class Parser:

    def parse(self, source_code):
        # Converts source code into abstract syntax tree (AST)
        ast_tree = generate_ast(source_code)
        return ast_tree

class ExecutionEngine:
```



```
    def execute(self, compiled_bytecode):
        # Executes bytecode in a virtual machine
        result = virtual_machine.execute_bytecode(compiled_bytecode)
        return result

class MemoryManager:

    def allocate(self, size):
        # Allocates memory for Python objects
        memory = allocate_memory(size)
        return memory

class ObjectModel:

    def __init__(self):
        # Defines the structure and behavior of Python objects
        pass

class GarbageCollector:

    def collect(self, memory):
        # Frees memory of unused objects
        garbage = identify_garbage(memory)
        deallocate(garbage)
```

This Python-style pseudocode describes the main components of Python's architecture:

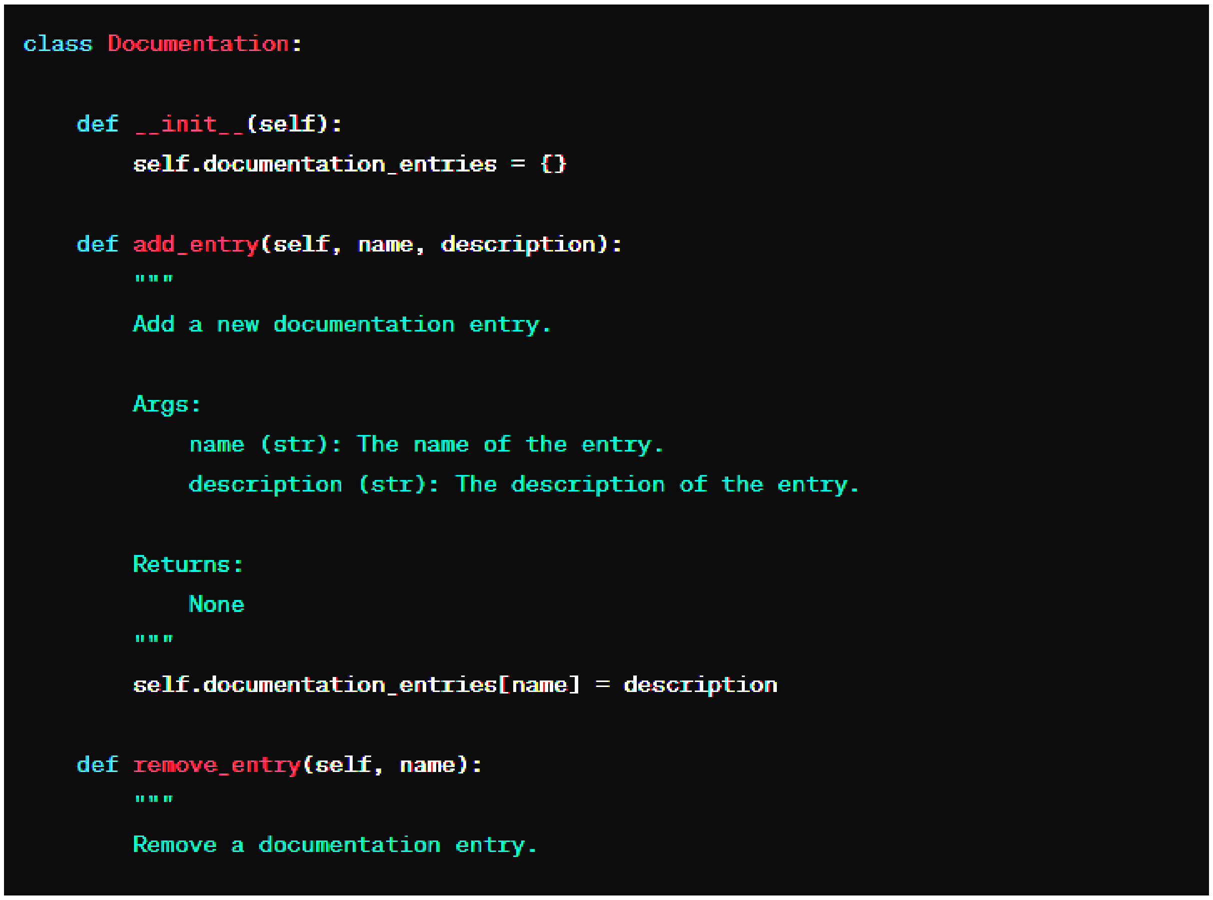
- Interpreter: Responsible for executing Python code by coordinating different components like parsing, compiling, and executing.
- Standard Library: Contains built-in modules and functions that provide a wide range of functionalities.
- Bytecode Compiler: Translates parsed Python code into bytecode, which is then executed by the virtual machine.
- Parser: Converts source code into an abstract syntax tree (AST) for further processing.
- Execution Engine: Executes bytecode within a virtual machine environment.

- Memory Manager: Allocates and manages memory for Python objects.
- Object Model: Defines the structure and behavior of Python objects.
- Garbage Collector: Identifies and frees memory occupied by unused objects to prevent memory leaks.

This architecture demonstrates how Python processes code from source to execution, handling memory management and providing a rich set of functionalities through its standard library and object model.

DOCUMENTATION IN PYTHON

Certainly! Below is a Python-style pseudocode description of how you might implement documentation in Python:



```python
class Documentation:

    def __init__(self):
        self.documentation_entries = {}

    def add_entry(self, name, description):
        """
        Add a new documentation entry.

        Args:
            name (str): The name of the entry.
            description (str): The description of the entry.

        Returns:
            None
        """
        self.documentation_entries[name] = description

    def remove_entry(self, name):
        """
        Remove a documentation entry.
        Args:
            name (str): The name of the entry to be removed.

        Returns:
            None
        """
        if name in self.documentation_entries:
            del self.documentation_entries[name]

    def get_entry(self, name):
        """
        Retrieve the description of a documentation entry.

        Args:
            name (str): The name of the entry to retrieve.

        Returns:
            str: The description of the entry.
        """
        return self.documentation_entries.get(name, "Entry not found")
```

```
# Example Usage:
doc = Documentation()

# Add entries
doc.add_entry("function1", "This is the documentation for function1")
doc.add_entry("class1", "This is the documentation for class1")

# Get entry
print(doc.get_entry("function1"))  # Output: This is the documentation for function1

# Remove entry
doc.remove_entry("class1")

# Trying to get removed entry
print(doc.get_entry("class1"))  # Output: Entry not found
```

In this pseudocode:

- The ‘Documentation’ class represents a collection of documentation entries.
- Each entry consists of a name and a description.
- The ‘add_entry’ method allows adding new documentation entries.

- The ‘remove_entry’ method allows removing documentation entries.
- The ‘get_entry’ method allows retrieving the description of a documentation entry by its name.

This structure provides a basic framework for managing documentation within a Python program. Actual Python documentation frameworks like Sphinx or tools like docstrings provide more comprehensive features and formatting options for documentation.