Python Syllabus: Sipalaya InfoTech

INTRODUCTION

- ø Python Overview
- ø Installation & Getting Started
- ø What is Syntax?
- ø Python Comments
- ø Python Variables

Python Data Types & Operators

- ø Data Types
- ø Python Numbers
- ø Data Conversion
- ø Type Casting
- ø Python Operators
- ø Python Booleans

Python Strings

- ø Python Strings
- ø Operation on Strings
- ø String Methods

- ø Format Strings
- ø Escape Characters

Python Lists

- ø Python Lists
- ø List Indexes
- ø Add List Items
- ø Remove List Items
- ø Change List Items
- ø List Comprehension
- ø List Methods

Python Tuples

- ø Python Tuples
- ø Tuple Indexes
- ø Manipulating Tuples
- ø Unpack Tuples

Python Sets

- ø Python Sets
- ø Add/Remove Set Items
- ø Join Sets

- ø Set Methods

Python Dictionaries

- ø Python Dictionaries
- ø Access Items
- ø Add/Remove Items
- ø Copy Dictionaries

Conditional Statements

- ø if Statement
- ø if-else Statement
- ø elif Statement
- ø Nested if Statement

Python Loops

- ø Python for Loop
- ø Python while Loop
- ø Nested Loops
- ø Control Statements

Python Functions

- ø Python Functions

- ø Function Arguments
- ø return Statement
- ø Python Recursion

Python Modules

- ø Python Modules
- ø Python Packages

Python OOPS

- ø Python OOPS
- ø self method
- ø __init__ method

Advanced Topics

- ø Python Iterators
- ø JSON
- ø Python try...except
- ø Python PIP
- ø Data & Time

File Handling

- ø Python File Handling
- ø Read/Write Files

Python Overview

What is Python

- Python is a dynamically typed, General Purpose Programming Language that supports an object-oriented programming approach as well as a functional programming approach.
- Python is also an interpreted and high-level programming language.
- It was created by Guido Van Rossum in 1989.

Features of Python

- Python is simple and easy to understand.
- It is Interpreted and platform-independent which makes debugging very easy.
- Python is an open-source programming language.
- Python provides very big library support. Some of the popular libraries include NumPy, Tensorflow, Selenium, OpenCV, etc.
- It is possible to integrate other programming languages within python.

What is Python used for

- Python is used in Data Visualization to create plots and graphical representations.
- Python helps in Data Analytics to analyze and understand raw data for insights and trends.
- It is used in AI and Machine Learning to simulate human behavior and to learn from past data without hard coding.
- It is used to create web applications.
- It can be used to handle databases.

- It is used in business and accounting to perform complex mathematical operations along with quantitative and qualitative analysis.

Installation & Getting Started

Steps to Install Python:

a. Visit the official python website: https://www.python.org/
b. Download the executable file based on your Operating System and version specifications.
c. Run the executable file and complete the installation process.

Version:

After installation check the version of python by typing following command: ‘python --version’.

Starting Python:

Open Python IDE or any other text editor of your preferred choice. Let’s understand python code execution with the simplest print statement. Type the following in the IDE:

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

Now save the file with a .py extension and Run it. You will get the following output:

```
Hello World !!!
```

Installing Packages:

To install packages in Python, we use the pip command.
e.g. pip install "Package Name"
Following command installs pandas package in Python

```
pip install pandas
```

We will learn more about the pip command in the chapter dedicated to pip.

What is Syntax?

In simplest words, *Syntax is the arrangement of words and phrases to create well-formed sentences in a language*. In the case of a computer language, the syntax is the structural arrangement of comments, variables, numbers, operators, statements, loops, functions, classes, objects, etc. which helps us understand the meaning or semantics of a computer language.

For E.g. a ‘comment’ is used to explain the functioning of a block of code. It starts with a ‘#’.
More on comments in the comments chapter.

For E.g. a block of code is identified by an ‘indentation’. Have a look at the following code, here print(i) is said to be indented with respect to the link above it. In simple words, indentation is the addition of spaces before the line "print(i)"

```
for i in range(5):
    print(i)
```

Python Comments

A comment is a part of the coding file that the programmer does not want to execute, rather the programmer uses it to either explain a block of code or to avoid the execution of a specific part of code while testing.

Single-Line Comments:

To write a comment just add a '#' at the start of the line.

Example 1:

```
#This is a 'Single-Line Comment'
print("This is a print statement.")
```

Output:

```
This is a print statement
```

Example 2:

```
print("Hello World !!!") #Printing Hello World
```

Output:

```
Hello World !!!
```

Example 3:

```
print("Python Program")
#print("Python Program")
```

Output:

```
Python Program
```

Multi-Line Comments:

To write multi-line comments you can use ‘#’ at each line or you can use the multiline string.

Example 1: The use of ‘#’.

```
#It will execute a block of code if a specified condition is true.
#If the codition is false than it will execute another block of code.
p = 7
if (p > 5):
    print("p is greater than 5.")
else:
    print("p is not greater than 5.")
```

Output:

```
p is greater than 5.
```

Example 2: The use ultiline string.of m

```
"""This is an if-else statement.
It will execute a block of code if a specified condition is true.
If the condition is false then it will execute another block of
code."""
p = 7
if (p > 5):
    print("p is greater than 5.")
else:
    print("p is not greater than 5.")
```

Output:

```
p is greater than 5.
```

Python Variables

Variables are containers that store information that can be manipulated and referenced later by the programmer within the code.
In python, the programmer does not need to declare the variable type explicitly, we just need to assign the value to the variable.
Example:

```
name = "Abhishek"    #type str
age = 20             #type int
passed = True        #type bool
```

It is always advisable to keep variable names descriptive and to follow a set of conventions while creating variables:

- Variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable name must start with a letter or the underscore character.
- Variables are case sensitive.
- Variable name cannot start with a number.

Example:

```
Color = "yellow"     #valid variable name
cOlor = "red"        #valid variable name
_color = "blue"      #valid variable name

5color = "green"     #invalid variable name
$color = "orange"    #invalid variable name
```

Sometimes, a multi-word variable name can be difficult to read by the reader. To make it more readable, the programmer can do the following:
Example:

```
NameOfCity = "Mumbai"        #Pascal case
nameOfCity = "Berlin"        #Camel case
name_of_city = "Moscow"      #snake case
```

Scope of variable:

The scope of the variable is the area within which the variable has been created. Based on this a variable can either have a local scope or a global scope.

Local Variable:

A local variable is created within a function and can be only used inside that function. Such a variable has a local scope.

Example:

```
icecream = "Vanilla"     #global variable
def foods():
    vegetable = "Potato"     #local variable
    fruit = "Lichi"          #local variable
    print(vegetable + " is a local variable value.")
    print(icecream + " is a global variable value.")
    print(fruit + " is a local variable value.")

foods()
```

Output:

```
Potato is a local variable value.
Vanilla is a global variable value.
Lichi is a local variable value.
```

Global Variable:

A global variable is created in the main body of the code and can be used anywhere within the code. Such a variable has a global scope.

Example:

```
icecream = "Vanilla"     #global variable
def foods():
    vegetable = "Potato"     #local variable
    fruit = "Lichi"          #local variable
    print(vegetable + " is a local variable value.")

foods()
print(icecream + " is a global variable value.")
```

```
print(fruit + " is a local variable value.")
```

Output:

```
Potato is a local variable value.
Vanilla is a global variable value.
```

Data Types

Data type specifies the type of value a variable requires to do various operations without causing an error. By default, python provides the following built-in data types:

Numeric data: int, float, complex

int: 3, -8, 0
float: 7.349, -9.0, 0.0000001
complex: 6 + 2i
more on numeric data types in the number chapter.

Text data: str

str: “Hello World!!!”, “Python Programming”

Boolean data:

Boolean data consists of values True or False.

Sequenced data: list, tuple, range

list: A list is an ordered collection of data with elements separated by a comma and enclosed within square brackets. Lists are mutable and can be modified after creation.
Example:

```
list1 = [8, 2.3, [-4, 5], ["apple", "banana"]]

print(list1)
```

Output:

```
[8, 2.3, [-4, 5], ['apple', 'banana']
```

tuple: A tuple is an ordered collection of data with elements separated by a comma and enclosed within parentheses. Tuples are immutable and can not be modified after creation.
Example:

```
tuple1 = (("parrot", "sparrow"), ("Lion", "Tiger"))
print(tuple1)
```

Output:

```
(('parrot', 'sparrow'), ('Lion', 'Tiger'))
```

range: returns a sequence of numbers as specified by the user. If not specified by the user then it starts from 0 by default and increments by 1.
Example:

```
sequence1 = range(4,14,2)
for i in sequence1:
    print(i)
```

Output:

```
4
6
8
10
12
```

Mapped data: dict

dict: a dictionary is an unordered collection of data containing a key:value pair. The key:value pairs are enclosed within curly brackets.
Example:

```
dict1 = {"name":"Sakshi", "age":20, "canVote":True}
print(dict1)
```

Output:

```
{'name': 'Sakshi', 'age': 20, 'canVote': True}
```

Binary data: bytes, bytearray, memoryview

bytes: bytes() function is used to convert objects into byte objects, or create empty bytes object of the specified size.
Example:

```
#Converting string to bytes
str1 = "This is a string"
arr1 = bytes(str1, 'utf-8')
print(arr1)
arr2 = bytes(str1, 'utf-16')
print(arr2)

#Creating bytes of given size
bytestr = bytes(4)

print(bytestr)
```

Output:

```
b'This is a string'
b'\xff\xfeT\x00h\x00i\x00s\x00 \x00i\x00s\x00 \x00a\x00
\x00s\x00t\x00r\x00i\x00n\x00g\x00'

b'\x00\x00\x00\x00'
```

bytearray: bytearray() function is used to convert objects into bytearray objects, or create empty bytearray object of the specified size.
Example:

```
#Converting string to bytes
str1 = "This is a string"
arr1 = bytearray(str1, 'utf-8')
print(arr1)
arr2 = bytearray(str1, 'utf-16')
print(arr2)

#Creating bytes of given size
bytestr = bytearray(4)

print(bytestr)
```

Output:

```
bytearray(b'This is a string')
bytearray(b'\xff\xfeT\x00h\x00i\x00s\x00 \x00i\x00s\x00 \x00a\x00
\x00s\x00t\x00r\x00i\x00n\x00g\x00')
bytearray(b'\x00\x00\x00\x00')
```

memoryview: memoryview() function returns a memory view object from a specified object.
Example:

```
str1 = bytes("home", "utf-8")
memoryviewstr = memoryview(str1)
print(list(memoryviewstr[0:]))
```

Output:

```
[104, 111, 109, 101]
```

Set data:

Set is an unordered collection of elements in which no element is repeated. The elements of sets are separated by a comma and contained within curly braces.
Example:

```
set1 = {4, -5, 8, 3, 2.9}
print(set1)
```

Output:

```
{2.9, 3, 4, 8, -5}
```

None:

None is used to define a null value. When we assign a None value to a variable, we are essentially resetting it to its original empty state which is not the same as zero, an empty string or a False value.
Example:

```
state = None
print(type(state))
```

Output:

```
<class 'NoneType'>
```

Python Numbers

In Python, numbers are of the following data types:

- int
- float
- complex

int

int is a positive or a negative integer of any length. int should not be a decimal or a fraction.

Example:

```
int1 = -2345698
int2 = 0
int3 = 100548

print(type(int1))
print(type(int2))
print(type(int3))
```

Output:

```
<class 'int'>
<class 'int'>
<class 'int'>
```

Float

A float is a positive or a negative decimal number. It can be an exponential number or a fraction.

Example:

```
flt1 = -8.35245      #decimal number
flt2 = 0.000001      #decimal number
flt3 = 2.6E44        #exponential number
flt4 = -6.022e23     #exponential number
```

```
print(type(flt1))
print(type(flt2))
print(type(flt3))

print(type(flt4))
```

Output:

```
<class 'float'>
<class 'float'>
<class 'float'>

<class 'float'>
```

complex

Complex numbers are a combination of real and imaginary number. They are of the form a + bj, where a is the real part and bj is the imaginary part.
Example:

```
cmplx1 = 2 + 4j
cmplx2 = -(3 + 7j)
cmplx3 = -4.1j
cmplx4 = 6j

print(type(cmplx1))
print(type(cmplx2))
print(type(cmplx3))

print(type(cmplx4))
```

Output:

```
<class 'complex'>
<class 'complex'>
<class 'complex'>

<class 'complex'>
```

Data Conversion

- The process of converting numeric data from one type to another is called as type conversion.
- To convert from integer to float, we use float() function.

To convert from integer to complex, we use the complex() function.

Example:

```
num1 = -25
num2 = float(num1)
num3 = complex(num1)

print(num2)
print(num3)
```

Output:

```
-25.0
(-25+0j)
```

- To convert from float to integer, we use int() function. int() function rounds of the given number to the nearest integer value.

To convert from float to complex, we use the complex() function.

Example:

```
num1 = 8.4
num2 = int(num1)
num3 = complex(num1)

print(num2)
print(num3)
```

Output:

```
8
(8.4+0j)
```

Note: complex numbers cannot be converted to integer or float.

Type Casting

Similar to type conversion, type casting is done when we want to specify a type on a variable.

Example:

```
str1 = "7"
str2 = "3.142"
str3 = "13"
num1 = 29
num2 = 6.67

print(int(str1))
print(float(str2))
print(float(str3))
print(str(num1))
print(str(num2))
```

Output:

```
7
3.142
13.0
29
6.67
```


Python Operators

Python has different types of operators for different operations. They are as follows:

Arithmetic operators:

Arithmetic operators are used to perform arithmetic/mathematical operations.

Name	Operator	Example
Addition	+	a+b
Subtraction	-	a-b
Multiplication	*	a*b
Division	/	a/b
Exponential	**	a**b
Modulus	%	a%b
Floor Division	//	a//b

Assignment operators:

These operators are used to assign values to variables.

Name	Evaluated As
=	a=b
+=	a+=b or a=a+b
-=	a-=b or a=a-b
*=	a*=b or a=a*b
**=	a**=b or a=a**b
/*	a/=b or a=a/b
//=	a//=b or a=a//b
%=	a%=b or a=a%b
&=	a&=b or a=a&b

|=	a|=b or a=a|b
^=	a^=b or a=a^b
>>=	a>>=b or a=a>>b
<<=	a<<=b or a=a<<b

Bitwise operators:

Bitwise operators are used to deal with binary operations.

Name	Operator	Example
Bitwise AND	Bitwise AND	a & b
Bitwise OR	Bitwise OR	a | b
Bitwise NOT	Bitwise NOT	~a
Bitwise XOR	Bitwise XOR	a ^ b
Bitwise right shift	Bitwise right shift	a>>
Bitwise left shift	Bitwise left shift	b<<

Comparison operators:

These operators are used to compare values.

Name	Operator	Example
Equal	==	a==b
Not Equal	!=	a!=b
Less Than	<	a>b
Greater Than	>	a<b
Less Than or Equal to	<=	a>=b
Greater Than or Equal to	>=	a<=b

Identity operators:

Name	Example	Evaluated As
is	a is b	Returns True if a and b are same
is not	a is not b	Returns True if a and b are not same

Logical operators:

These operators are used to deal with logical operations.

Name	Operator	Example
AND	and	a=2 and b=3
OR	or	a=2 or b=3
NOT	not	Not(a=2 or b=3)

Membership operators:

Name	Example	Evaluated As
in	a in b	Returns True if a is present in given sequence or collection
not in	a not in b	Returns True if a is not present in given sequence or collection

Operator Precedence in Python:

Name	Operator
Parenthesis	()
Exponential	**
Complement, unary plus, unary minus	~ , +, -
Multiply, divide, modulus, floor division	*, /, %, //

Addition, subtraction	+, -
Left shift and right shift operators	<<, >>
Bitwise and	&
Bitwise or and xor	^, |
Comparison operators	<, >, >=, <=
Equality operators	==, !=
Assignment operators	=, %=, /=, //=, -=, +=, *= , **=
Identity operators	is, is not
Membership operators	in, not in
Logical operators	and, or, not

Python Booleans

Boolean consists of only two values; True and False.

Why are Boolean's needed?

Consider the following if-else statement:

```
x = 13
if(x>13):
    print("X is a prime number.")
else:
    print("X is not a prime number.")
```

Is it True that X is greater than 13 or is it False?

- Thus Booleans are used to know whether the given expression is True or False.
- bool() function evaluates values and returns True or False.

Here are some examples where the Boolean returns True/False values for different datatypes.

None:

```
print("None: ",bool(None))
```

Output:

```
None:  False
```

Numbers:

```
print("Zero:",bool(0))
print("Integer:",bool(23))
print("Float:",bool(3.142))
print("Complex:",bool(5+2j))
```

Output:

```
Zero: False
Integer: True
```

```
Float: True
Complex: True
```

Strings:

```
#Strings
print("Any string:",bool("Nilesh"))
print("A string containing number:",bool("8.5"))
print("Empty string:" ,"")
```

Output:

```
Any string: True
A string containing number: True
Empty string: False
```

Lists:

```
print("Empty List:",bool([]))
print("List:",bool([1,2,5,2,1,3]))
```

Output:

```
Empty List: False
List: True
```

Tuples:

```
#Tuples
print("Empty Tuple:",bool(()))
print("Tuple:",bool(("Horse", "Rhino", "Tiger")))
```

Output:

```
Empty Tuple: False
Tuple: True
```

Sets and Dictionaries:

```
print("Empty Dictionary:",bool({}))
print("Empty Set:",bool({"Mike", 22, "Science"}))
print("Dictionary:",bool({"name":"Lakshmi", "age":24
,"job":"unemployed"}))
```

Output:

```
Empty Dictionary: False
Empty Set: True
Dictionary: True
```


Python Strings

What are strings?

In python, anything that you enclose between single or double quotation marks is considered a string. A string is essentially a sequence or array of textual data.
Strings are used when working with Unicode characters.
Example:

```
name = "Samuel"

print("Hello, " + name)
```

Output:

```
Hello, Samuel
```

Note: It does not matter whether you enclose your strings in single or double quotes, the output remains the same.

Sometimes, the user might need to put quotation marks in between the strings. Example, consider the sentence: He said, “I want to eat an apple”. How will you print this statement in python?

Wrong way ❌

```
print("He said, "I want to eat an apple".")
```

Output:

```
print("He said, "I want to eat an apple".")
                          ^
SyntaxError: invalid syntax
```

Right way ✔️

```
print('He said, "I want to eat an apple".')
#OR
```

```
print("He said, \"I want to eat an apple\".")
```

Output:

```
He said, "I want to eat an apple".
```

```
He said, "I want to eat an apple".
```

What if you want to write multiline strings?

Sometimes the programmer might want to include a note, or a set of instructions, or just might want to explain a piece of code. Although this might be done using multiline comments, the programmer may want to display this in the execution of programmer. This is done using multiline strings.

Example:

```
receipe = """
1. Heat the pan and add oil
2. Crack the egg
3. Add salt in egg and mix well
4. Pour the mixture in pan
5. Fry on medium heat
"""
print(receipe)

note = '''
This is a multiline string
It is used to display multiline message in the program
'''

print(note)
```

Output:

```
1. Heat the pan and add oil
2. Crack the egg
3. Add salt in egg and mix well
4. Pour the mixture in pan
5. Fry on medium heat
```

```
This is a multiline string
It is used to display multiline message in the program
```

Operation on Strings

Length of a String:

We can find the length of a string using len() function.
Example:

```
fruit = "Mango"
len1 = len(fruit)
print("Mango is a", len1, "letter word.")
```

Output:

```
Mango is a 5 letter word.
```

String as an Array:

A string is essentially a sequence of characters also called an array. Thus we can access the elements of this array.
Example:

```
pie = "ApplePie"
print(pie[:5])
print(pie[6])    #returns character at specified index
```

Output:

```
Apple
i
```

Note: This method of specifying the start and end index to specify a part of a string is called slicing.
Example:

```
pie = "ApplePie"
print(pie[:5])       #Slicing from Start
print(pie[5:])       #Slicing till End
print(pie[2:6])      #Slicing in between
```

```
print(pie[-8:])      #Slicing using negative index
```

Output:

```
Apple
Pie
pleP

ApplePie
```

Loop through a String:

Strings are arrays and arrays are iterable. Thus we can loop through strings.

Example:

```
alphabets = "ABCDE"
for i in alphabets:

    print(i)
```

Output:

```
A
B
C
D

E
```

String Methods

Python provides a set of built-in methods that we can use to alter and modify the strings.

upper() : The upper() method converts a string to upper case.

Example:

```
str1 = "AbcDEfghIJ"
print(str1.upper())
```

Output:

```
ABCDEFGHIJ
```

lower() : The lower() method converts a string to upper case.

Example:

```
str1 = "AbcDEfghIJ"
print(str1.lower())
```

Output:

```
abcdefghij
```

strip() : The strip() method removes any white spaces before and after the string.

Example:

```
str2 = " Silver Spoon "
print(str2.strip)
```

Output:

```
Silver Spoon
```

rstrip() : the rstrip() removes any trailing characters.

Example:

```
str3 = "Hello !!!"
print(str3.rstrip("!"))
```

Output:

```
Hello
```

replace() : the replace() method replaces a string with another string.

Example:

```
str2 = "Silver Spoon"
print(str2.replace("Sp", "M"))
```

Output:

```
Silver Moon
```

split() : The split() method splits the give string at the specified instance and returns the separated strings as list items.

Example:

```
str2 = "Silver Spoon"
print(str2.split(" "))      #Splits the string at the whitespace " ".
```

Output:

```
['Silver', 'Spoon']
```

There are various other string methods that we can use to modify our strings.

capitalize() : The capitalize() method turns only the first character of the string to uppercase and the rest other characters of the string are

turned to lowercase. The string has no effect if the first character is already uppercase.

Example:

```
str1 = "hello"
capStr1 = str1.capitalize()
print(capStr1)
str2 = "hello WorlD"
capStr2 = str2.capitalize()
print(capStr2)
```

Output:

```
Hello
Hello world
```

center() : The center() method aligns the string to the center as per the parameters given by the user.

Example:

```
str1 = "Welcome to the Console!!!"
print(str1.center(50))
```

Output:

```
          Welcome to the Console!!!
```

We can also provide padding character. It will fill the rest of the fill characters provided by the user.

Example:

```
str1 = "Welcome to the Console!!!"
print(str1.center(50, "."))
```

Output:

```
............Welcome to the Console!!!.............
```

count() : The count() method returns the number of times the given value has occurred within the given string.

Example:

```
str2 = "Abracadabra"
countStr = str2.count("a")
print(countStr)
```

Output:

```
4
```

endswith() : The endswith() method checks if the string ends with a given value. If yes then return True, else return False.

Example 1:

```
str1 = "Welcome to the Console !!!"
print(str1.endswith("!!!"))
```

Output:

```
True
```

Example 2:

```
str1 = "Welcome to the Console !!!"
print(str1.endswith("Console"))
```

Output:

```
False
```

We can even also check for a value in-between the string by providing start and end index positions.

Example:

```
str1 = "Welcome to the Console !!!"
print(str1.endswith("to", 4, 10))
```

Output:

```
True
```

find() : The find() method searches for the first occurrence of the given value and returns the index where it is present. If given value is absent from the string then return -1.

Example:

```
str1 = "He's name is Dan. He is an honest man."
print(str1.find("is"))
```

Output:

```
10
```

As we can see, this method is somewhat similar to the index() method. The major difference being that index() raises an exception if value is absent whereas find() does not.

Example:

```
str1 = "He's name is Dan. He is an honest man."
print(str1.find("Daniel"))
```

Output:

```
-1
```

index() : The index() method searches for the first occurrence of the given value and returns the index where it is present. If given value is absent from the string then raise an exception.

Example:

```
str1 = "He's name is Dan. Dan is an honest man."
print(str1.index("Dan"))
```

Output:

```
13
```

As we can see, this method is somewhat similar to the find() method. The major difference being that index() raises an exception if value is absent whereas find() does not.

Example:

```
str1 = "He's name is Dan. Dan is an honest man."
print(str1.index("Daniel"))
```

Output:

```
ValueError: substring not found
```

isalnum() : The isalnum() method returns True only if the entire string only consists of A-Z, a-z, 0-9. If any other characters or punctuations are present, then it returns False.

Example 1:

```
str1 = "WelcomeToTheConsole"
print(str1.isalnum())
```

Output:

```
True
```

Example 2:

```
str1 = "Welcome To The Console"
print(str1.isalnum())
str2 = "Hurray!!!"
print(str2.isalnum())
```

Output:

```
False
False
```

isalpha() : The isalnum() method returns True only if the entire string only consists of A-Z, a-z. If any other characters or punctuations or numbers(0-9) are present, then it returns False.

Example 1:

```
str1 = "Welcome"
print(str1.isalpha())
```

Output:

```
True
```

Example 2:

```
tr1 = "I'm 18 years old"
print(str1.isalpha())
str2 = "Hurray!!!"
print(str2.isalnum())
```

Output:

```
False
False
```

islower() : The islower() method returns True if all the characters in the string are lower case, else it returns False.

Example 1:

```
str1 = "hello world"
print(str1.islower())
```

Output:

```
True
```

Example 2:

```
str1 = "welcome Mike"
print(str1.islower())
str2 = "Hurray!!!"
print(str2.islower())
```

Output:

```
False
```

```
False
```

isprintable() : The isprintable() method returns True if all the values within the given string are printable, if not, then return False.

Example 1:

```
str1 = "We wish you a Merry Christmas"
print(str1.isprintable())
```

Output:

```
True
```

Example 2:

```
str2 = "Hello, \t\t.Mike"
print(str2.isprintable())
```

Output:

```
False
```

isspace() : The isspace() method returns True only and only if the string contains white spaces, else returns False.

Example 1:

```
str1 = "          "       #using Spacebar
print(str1.isspace())
str2 = "          "       #using Tab
print(str2.isspace())
```

Output:

```
True
True
```

Example 2:

```
str1 = "Hello World"
```

```
print(str1.isspace())
```

Output:

```
False
```

istitle() : The istitile() returns True only if the first letter of each word of the string is capitalized, else it returns False.

Example 1:

```
str1 = "World Health Organization"
print(str1.istitle())
```

Output:

```
True
```

Example 2:

```
str2 = "To kill a Mocking bird"
print(str2.istitle())
```

Output:

```
False
```

isupper() : The isupper() method returns True if all the characters in the string are upper case, else it returns False.

Example 1:

```
str1 = "WORLD HEALTH ORGANIZATION"
print(str1.isupper())
```

Output:

```
True
```

Example 2:

```
str2 = "To kill a Mocking bird"
```

```
print(str2.isupper())
```

Output:

```
False
```

replace() : The replace() method can be used to replace a part of the original string with another string.

Example:

```
str1 = "Python is a Compiled Language."
print(str1.replace("Compiled", "Interpreted"))
```

Output:

```
Python is a Interpreted Language.
```

startswith() : The endswith() method checks if the string starts with a given value. If yes then return True, else return False.

Example 1:

```
str1 = "Python is a Interpreted Language"
print(str1.startswith("Python"))
```

Output:

```
True
```

Example 2:

```
str1 = "Python is a Interpreted Language"
print(str1.startswith("a"))
```

Output:

```
False
```

We can even also check for a value in-between the string by providing start and end index positions.

Example:

```
str1 = "Python is a Interpreted Language"
print(str1.startswith("Inter", 12, 20))
```

Output:

```
True
```

swapcase() : The swapcase() method changes the character casing of the string. Upper case are converted to lower case and lower case to upper case.

Example:

```
str1 = "Python is a Interpreted Language"
print(str1.swapcase())
```

Output:

```
pYTHON IS A iNTERPRETED lANGUAGE
```

title() : The title() method capitalizes each letter of the word within the string.

Example:

```
str1 = "He's name is Dan. Dan is an honest man."
print(str1.title())
```

Output:

```
He'S Name Is Dan. Dan Is An Honest Man.
```

Format Strings

What if we want to join two separated strings?

We can perform concatenation to join two or more separate strings.
Example:

```
str4 = "Captain"
str5 = "America"
str6 = str4 + " " + str5

print(str6)
```

Output:

```
Captain America
```

In the above example, we saw how one can concatenate two strings. But how can we concatenate a string and an integer?

```
name = "Guzman"
age = 18

print("My name is" + name + "and my age is" + age)
```

Output:

```
TypeError: can only concatenate str (not "int") to str
```

As we can see, we cannot concatenate a string to another data type.

So what's the solution?

We make the use of format() method. The format() methods places the arguments within the string wherever the placeholders are specified.
Example:

```
name = "Guzman"
age = 18
```

```
statement = "My name is {} and my age is {}."

print(statement.format(name, age))
```

Output:

```
My name is Guzman and my age is 18.
```

We can also make the use of indexes to place the arguments in specified order.

Example:

```
quantity = 2
fruit = "Apples"
cost = 120.0
statement = "I want to buy {2} dozen {0} for {1}$"

print(statement.format(fruit,cost,quantity))
```

Output:

```
I want to buy 2 dozen Apples for $120.0
```

Escape Characters

Escape Characters are very important in python. It allows us to insert illegal characters into a string like a back slash, new line or a tab.
Single/Double Quote: used to insert single/double quotes in the string.
Example:

```
str1 = "He was \"Flabergasted\"."
str2 = 'He was \'Flabergasted\'.'
print(str1)
print(str2)
```

Output:

```
He was "Flabergasted".
He was 'Flabergasted'.
```

New Line: inserts a new line wherever specified.

Example:

```
str1 = "I love doing Yoga. \nIt cleanses my mind and my body."
print(str1)
```

Output:

```
I love doing Yoga.
It cleanses my mind and my body.
```

Tab: inserts a tab wherever specified.

Example:

```
str2 = "Hello \t\tWorld \t!!!"
print(str2)
```

Output:

```
Hello           World   !!!
```

Backspace: erases the character before it wherever it is specified.

Example:

```
str2 = "Hello  \bWorld !!!"
print(str2)
```

Output:

```
Hello World !!!
```

Backslash: used to insert a backslash into a string.

Example:

```
str3 = "What will you eat? Apple\\Banana"
print(str3)
```

Output:

```
What will you eat? Apple\Banana
```

Python Lists

- Lists are ordered collection of data items.
- They store multiple items in a single variable.
- List items are separated by commas and enclosed within square brackets [].
- Lists are changeable meaning we can alter them after creation.

Example 1:

```
lst1 = [1,2,2,3,5,4,6]
lst2 = ["Red", "Green", "Blue"]
print(lst1)
print(lst2)
```

Output:

```
[1, 2, 2, 3, 5, 4, 6]
['Red', 'Green', 'Blue']
```

Example 2:

```
details = ["Abhijeet", 18, "FYBScIT", 9.8]
print(details)
```

Output:

```
['Abhijeet', 18, 'FYBScIT', 9.8]
```

As we can see, a single list can contain items of different datatypes.

List Indexes

Each item/element in a list has its own unique index. This index can be used to access any particular item from the list. The first item has index [0], second item has index [1], third item has index [2] and so on.
Example:

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]

#          [0]      [1]      [2]      [3]      [4]
```

Accessing list items:

Positive Indexing:

As we have seen that list items have index, as such we can access items using these indexes.
Example:

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
#          [0]      [1]      [2]      [3]      [4]
print(colors[2])
print(colors[4])

print(colors[0])
```

Output:

```
Blue
Green

Red
```

Negative Indexing:

Similar to positive indexing, negative indexing is also used to access items, but from the end of the list. The last item has index [-1], second last item has index [-2], third last item has index [-3] and so on.
Example:

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
#          [-5]    [-4]    [-3]     [-2]      [-1]
print(colors[-1])
print(colors[-3])

print(colors[-5])
```

Output:

```
Green
Blue

Red
```

Check for item:

We can check if a given item is present in the list. This is done using the in keyword.

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
if "Yellow" in colors:
    print("Yellow is present.")
else:

    print("Yellow is absent.")
```

Output:

```
Yellow is present.
```

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
if "Orange" in colors:
    print("Orange is present.")
else:

    print("Orange is absent.")
```

Output:

```
Orange is absent.
```

Range of Index:

You can print a range of list items by specifying where do you want to start, where do you want to end and if you want to skip elements in between the range.
Syntax:
List[start : end : jumpIndex]
Note: jump Index is optional. We will see this in given examples.

Example: printing elements within a particular range:

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]
print(animals[3:7])   #using positive indexes
print(animals[-7:-2]) #using negative indexes
```

Output:

```
['mouse', 'pig', 'horse', 'donkey']
['bat', 'mouse', 'pig', 'horse', 'donkey']
```

Here, we provide index of the element from where we want to start and the index of the element till which we want to print the values.
Note: The element of the end index provided will not be included.

Example: printing all element from a given index till the end

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]
print(animals[4:])    #using positive indexes
print(animals[-4:])   #using negative indexes
```

Output:

```
['pig', 'horse', 'donkey', 'goat', 'cow']
['horse', 'donkey', 'goat', 'cow']
```

When no end index is provided, the interpreter prints all the values till the end.

Example: printing all elements from start to a given index

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]
print(animals[:6])    #using positive indexes
```

```
print(animals[:-3])   #using negative indexes
```

Output:

```
['cat', 'dog', 'bat', 'mouse', 'pig', 'horse']
```

```
['cat', 'dog', 'bat', 'mouse', 'pig', 'horse']
```

When no start index is provided, the interpreter prints all the values from start up to the end index provided.

Example: print alternate values

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey",
"goat", "cow"]
print(animals[::2])         #using positive indexes
```

```
print(animals[-8:-1:2])     #using negative indexes
```

Output:

```
['cat', 'bat', 'pig', 'donkey', 'cow']
```

```
['dog', 'mouse', 'horse', 'goat']
```

Here, we have not provided start and index, which means all the values will be considered. But as we have provided a jump index of 2 only alternate values will be printed.

Example: printing every 3rd consecutive withing given range

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey",
"goat", "cow"]
```

```
print(animals[1:8:3])
```

Output:

```
['dog', 'pig', 'goat']
```

Here, jump index is 3. Hence it prints every 3rd element within given index.

Add List Items

There are three methods to add items to list: append(), insert() and extend()

append():

This method appends items to the end of the existing list.
Example:

```
colors = ["voilet", "indigo", "blue"]
colors.append("green")

print(colors)
```

Output:

```
['voilet', 'indigo', 'blue', 'green']
```

What if you want to insert an item in the middle of the list? At a specific index?

insert():

This method inserts an item at the given index. User has to specify index and the item to be inserted within the insert() method.
Example:

```
colors = ["voilet", "indigo", "blue"]
#           [0]        [1]       [2]

colors.insert(1, "green")    #inserts item at index 1
# updated list: colors = ["voilet", "green", "indigo", "blue"]
#       indexs               [0]       [1]       [2]       [3]


print(colors)
```

Output:

```
['voilet', 'green', 'indigo', 'blue']
```

What if you want to append an entire list or any other collection (set, tuple, dictionary) to the existing list?

extend():

This method adds an entire list or any other collection datatype (set, tuple, dictionary) to the existing list.

Example 1:

```
#add a list to a list
colors = ["voilet", "indigo", "blue"]
rainbow = ["green", "yellow", "orange", "red"]
colors.extend(rainbow)

print(colors)
```

Output:

```
['voilet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']
```

Example 2:

```
#add a tuple to a list
cars = ["Hyundai", "Tata", "Mahindra"]
cars2 = ("Mercedes", "Volkswagen", "BMW")
cars.extend(cars2)

print(cars)
```

Output:

```
['Hyundai', 'Tata', 'Mahindra', 'Mercedes', 'Volkswagen', 'BMW']
```

Example 3:

```
#add a set to a list
cars = ["Hyundai", "Tata", "Mahindra"]
cars2 = {"Mercedes", "Volkswagen", "BMW"}
cars.extend(cars2)

print(cars)
```

Output:

```
['Hyundai', 'Tata', 'Mahindra', 'Mercedes', 'BMW', 'Volkswagen']
```

Example 4:

```
#add a dictionary to a list
students = ["Sakshi", "Aaditya", "Ritika"]
students2 = {"Yash":18, "Devika":19, "Soham":19}    #only add keys,
does not add values
students.extend(students2)

print(students)
```

Output:

```
['Sakshi', 'Aaditya', 'Ritika', 'Yash', 'Devika', 'Soham']
```

concatenate two lists:

you can simply concatenate two list to join two lists.

Example:

```
colors = ["voilet", "indigo", "blue", "green"]
colors2 = ["yellow", "orange", "red"]

print(colors + colors2)
```

Output:

```
['voilet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']
```

Remove List Items

There are various methods to remove items from the list: pop(), remove(), del(), clear()

pop():

This method removes the last item of the list if no index is provided. If an index is provided, then it removes item at that specified index.
Example 1:

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
colors.pop()          #removes the last item of the list
print(colors)
```

Output:

```
['Red', 'Green', 'Blue', 'Yellow']
```

Example 2:

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]
colors.pop(1)         #removes item at index 1
print(colors)
```

Output:

```
['Red', 'Blue', 'Yellow', 'Green']
```

What if you want to remove a specific item from the list?

remove():

This method removes specific item from the list.
Example:

```
colors = ["voilet", "indigo", "blue", "green", "yellow"]
colors.remove("blue")
print(colors)
```

Output:

```
['voilet', 'indigo', 'green', 'yellow']
```

del:

del is not a method, rather it is a keyword which deletes item at specific from the list, or deletes the list entirely.
Example 1:

```
colors = ["voilet", "indigo", "blue", "green", "yellow"]
del colors[3]
print(colors)
```

Output:

```
['voilet', 'indigo', 'blue', 'yellow']
```

Example 2:

```
colors = ["voilet", "indigo", "blue", "green", "yellow"]
del colors
print(colors)
```

Output:

```
NameError: name 'colors' is not defined
```

We get an error because our entire list has been deleted and there is no variable called colors which contains a list.

What if we don’t want to delete the entire list, we just want to delete all items within that list?

clear():

This method clears all items in the list and prints an empty list.
Example:

```
colors = ["voilet", "indigo", "blue", "green", "yellow"]
colors.clear()
print(colors)
```

Output:

```
[]
```

Change List Items

Changing items from list is easier, you just have to specify the index where you want to replace the item with existing item.
Example:

```
names = ["sipalaya", "Sarah", "Nadia", "Oleg", "Steve"]
names[2] = "Millie"
print(names)
```

Output:

```
['sipalaya', 'Sarah', 'Millie', 'Oleg', 'Steve']
```

You can also change more than a single item at once. To do this, just specify the index range over which you want to change the items.
Example:

```
names = ["sipalaya", "Sarah", "Nadia", "Oleg", "Steve"]
names[2:4] = ["juan", "Anastasia"]
print(names)
```

Output:

```
['sipalaya', 'Sarah', 'juan', 'Anastasia', 'Steve']
```

What if the range of the index is more than the list of items provided? In this case, all the items within the index range of the original list are replaced by the items that are provided.
Example:

```
names = ["sipalaya", "Sarah", "Nadia", "Oleg", "Steve"]
names[1:4] = ["juan", "Anastasia"]
print(names)
```

Output:

```
['sipalaya', 'juan', 'Anastasia', 'Steve']
```

What if we have more items to be replaced than the index range provided? In this case, the original items within the range are replaced by the new items and the remaining items move to the right of the list accordingly.
Example:

```
names = ["sipalaya", "Sarah", "Nadia", "Oleg", "Steve"]
names[2:3] = ["juan", "Anastasia", "Bruno", "Olga", "Rosa"]
print(names)
```

Output:

```
['sipalaya', 'Sarah', 'juan', 'Anastasia', 'Bruno', 'Olga', 'Rosa',
'Oleg', 'Steve']
```

List Comprehension

List comprehensions are used for creating new lists from other iterables like lists, tuples, dictionaries, sets, and even in arrays and strings.

Syntax:
List = [expression(item) for item in iterable if condition]
expression: it is the item which is being iterated.
iterable: it can be list, tuples, dictionaries, sets, and even in arrays and strings.
condition: condition checks if the item should be added to the new list or not.

Example 1: accepts items with the small letter “o” in the new list

```
names = ["Milo", "Sarah", "Bruno", "Anastasia", "Rosa"]
namesWith_O = [item for item in names if "o" in item]

print(namesWith_O)
```

Output:

```
['Milo', 'Bruno', 'Rosa']
```

Example 2: accepts items which have more than 4 letters

```
names = ["Milo", "Sarah", "Bruno", "Anastasia", "Rosa"]
namesWith_O = [item for item in names if (len(item) > 4)]

print(namesWith_O)
```

Output:

```
['Sarah', 'Bruno', 'Anastasia']
```

List Methods

We have discussed methods like append(), clear(), extend(), insert(), pop(), remove() before. Now we will learn about some more list methods:

sort(): This method sorts the list in ascending order.

Example 1:

```
colors = ["voilet", "indigo", "blue", "green"]
colors.sort()
print(colors)

num = [4,2,5,3,6,1,2,1,2,8,9,7]
num.sort()
print(num)
```

Output:

```
['blue', 'green', 'indigo', 'voilet']
[1, 1, 2, 2, 2, 3, 4, 5, 6, 7, 8, 9]
```

What if you want to print the list in descending order?

We must give reverse=True as a parameter in the sort method.

Example:

```
colors = ["voilet", "indigo", "blue", "green"]
colors.sort(reverse=True)
print(colors)

num = [4,2,5,3,6,1,2,1,2,8,9,7]
num.sort(reverse=True)
print(num)
```

Output:

```
['voilet', 'indigo', 'green', 'blue']
[9, 8, 7, 6, 5, 4, 3, 2, 2, 2, 1, 1]
```

The reverse parameter is set to False by default.
Note: Do not mistake the reverse parameter with the reverse method.

reverse(): This method reverses the order of the list.

Example:

```
colors = ["voilet", "indigo", "blue", "green"]
colors.reverse()
print(colors)

num = [4,2,5,3,6,1,2,1,2,8,9,7]
num.reverse()
print(num)
```

Output:

```
['green', 'blue', 'indigo', 'voilet']
[7, 9, 8, 2, 1, 2, 1, 6, 3, 5, 2, 4]
```

index(): This method returns the index of the first occurrence of the list item.

Example:

```
colors = ["voilet", "green", "indigo", "blue", "green"]
print(colors.index("green"))

num = [4,2,5,3,6,1,2,1,3,2,8,9,7]
print(num.index(3))
```

Output:

```
1
3
```

count(): Returns the count of the number of items with the given value.

Example:

```
colors = ["voilet", "green", "indigo", "blue", "green"]
print(colors.count("green"))

num = [4,2,5,3,6,1,2,1,3,2,8,9,7]
```

Output:

```
2
3
```

copy(): Returns copy of the list. This can be done to perform operations on the list without modifying the original list.

Example

```
colors = ["voilet", "green", "indigo", "blue"]
newlist = colors.copy()
print(colors)
print(newlist)
```

Output:

```
['voilet', 'green', 'indigo', 'blue']
['voilet', 'green', 'indigo', 'blue']
```

Python Tuples

Tuples are ordered collection of data items. They store multiple items in a single variable. Tuple items are separated by commas and enclosed within round brackets (). Tuples are unchangeable meaning we can not alter them after creation.

Example 1:

```
tuple1 = (1,2,2,3,5,4,6)
tuple2 = ("Red", "Green", "Blue")
print(tuple1)
print(tuple2)
```

Output:

```
(1, 2, 2, 3, 5, 4, 6)
('Red', 'Green', 'Blue')
```

Example 2:

```
details = ("Abhijeet", 18, "FYBScIT", 9.8)
print(details)
```

Output:

```
('Abhijeet', 18, 'FYBScIT', 9.8)
```

Tuple Indexes

Each item/element in a tuple has its own unique index. This index can be used to access any particular item from the tuple. The first item has index [0], second item has index [1], third item has index [2] and so on.
Example:

```
country = ("Spain", "Italy", "India", "England", "Germany")

#            [0]       [1]       [2]       [3]         [4]
```

Accessing tuple items:

I. Positive Indexing:

As we have seen that tuple items have index, as such we can access items using these indexes.
Example:

```
country = ("Spain", "Italy", "India", "England", "Germany")
#            [0]       [1]       [2]       [3]         [4]
print(country[1])
print(country[3])

print(country[0])
```

Output:

```
Italy
England

Spain
```

II. Negative Indexing:

Similar to positive indexing, negative indexing is also used to access items, but from the end of the tuple. The last item has index [-1], second last item has index [-2], third last item has index [-3] and so on.
Example:

```
country = ("Spain", "Italy", "India", "England", "Germany")
#            [0]       [1]       [2]        [3]         [4]
print(country[-1])
print(country[-3])

print(country[-4])
```

Output:

```
Germany
India

Italy
```

III. Check for item:

We can check if a given item is present in the tuple. This is done using the in keyword.

Example 1:

```
country = ("Spain", "Italy", "India", "England", "Germany")
if "Germany" in country:
    print("Germany is present.")
else:

    print("Germany is absent.")
```

Output:

```
Germany is present.
```

Example 2:

```
country = ("Spain", "Italy", "India", "England", "Germany")
if "Russia" in country:
    print("Russia is present.")
else:

    print("Russia is absent.")
```

Output:

```
Russia is absent.
```

IV. Range of Index:

You can print a range of tuple items by specifying where do you want to start, where do you want to end and if you want to skip elements in between the range.

Syntax:
Tuple[start : end : jumpIndex]
Note: jump Index is optional. We will see this in given examples.

Example: printing elements within a particular range:

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey",
"goat", "cow")
print(animals[3:7])      #using positive indexes
print(animals[-7:-2])    #using negative indexes
```

Output:

```
('mouse', 'pig', 'horse', 'donkey')
('bat', 'mouse', 'pig', 'horse', 'donkey')
```

Here, we provide index of the element from where we want to start and the index of the element till which we want to print the values.
Note: The element of the end index provided will not be included.

Example: printing all element from a given index till the end

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey",
"goat", "cow")
print(animals[4:])      #using positive indexes
print(animals[-4:])      #using negative indexes
```

Output:

```
('pig', 'horse', 'donkey', 'goat', 'cow')
('horse', 'donkey', 'goat', 'cow')
```

When no end index is provided, the interpreter prints all the values till the end.

Example: printing all elements from start to a given index

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey",
"goat", "cow")
```

```
print(animals[:6])       #using positive indexes
print(animals[:-3])      #using negative indexes
```

Output:

```
('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')
('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')
```

When no start index is provided, the interpreter prints all the values from start up to the end index provided.

Example: print alternate values

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey",
"goat", "cow")
print(animals[::2])      #using positive indexes
print(animals[-8:-1:2]) #using negative indexes
```

Output:

```
('cat', 'bat', 'pig', 'donkey', 'cow')
('dog', 'mouse', 'horse', 'goat')
```

Here, we have not provided start and index, which means all the values will be considered. But as we have provided a jump index of 2 only alternate values will be printed.

Example: printing every 3rd consecutive withing given range

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey",
"goat", "cow")
print(animals[1:8:3])
```

Output:

```
('dog', 'pig', 'goat')
```

Here, jump index is 3. Hence it prints every 3rd element within given index.

Manipulating Tuples

Tuples are immutable, hence if you want to add, remove or change tuple items, then first you must convert the tuple to a list. Then perform operation on that list and convert it back to tuple.

Example:

```
countries = ("Spain", "Italy", "India", "England", "Germany")
temp = list(countries)
temp.append("Russia")        #add item
temp.pop(3)                  #remove item
temp[2] = "Finland"          #change item
countries = tuple(temp)

print(countries)
```

Output:

```
('Spain', 'Italy', 'Finland', 'Germany', 'Russia')
```

Thus, we convert the tuple to a list, manipulate items of the list using list methods, then convert list back to a tuple.

However, we can directly concatenate two tuples instead of converting them to list and back.

Example:

```
countries = ("Pakistan", "Afghanistan", "Bangladesh", "ShriLanka")
countries2 = ("Vietnam", "India", "China")
southEastAsia = countries + countries2

print(southEastAsia)
```

Output:

```
('Pakistan', 'Afghanistan', 'Bangladesh', 'ShriLanka', 'Vietnam',
'India', 'China')
```

Unpack Tuples

Unpacking is the process of assigning the tuple items as values to variables.

Example:

```
info = ("Marcus", 20, "MIT")
(name, age, university) = info
print("Name:", name)
print("Age:",age)

print("Studies at:",university)
```

Output:

```
Name: Marcus
Age: 20

Studies at: MIT
```

Here, the number of list items is equal to the number of variables.

But what if we have more number of items then the variables?
You can add an * to one of the variables and depending upon the position of variable and number of items, python matches variables to values and assigns it to the variables.

Example 1:

```
fauna = ("cat", "dog", "horse", "pig", "parrot", "salmon")
(*animals, bird, fish) = fauna
print("Animals:", animals)
print("Bird:", bird)

print("Fish:", fish)
```

Output:

```
Animals: ['cat', 'dog', 'horse', 'pig']
Bird: parrot

Fish: salmon
```

Example 2:

```
fauna = ("parrot", "cat", "dog", "horse", "pig", "salmon")
(bird, *animals, fish) = fauna
print("Animals:", animals)
print("Bird:", bird)

print("Fish:", fish)
```

Output:

```
Animals: ['cat', 'dog', 'horse', 'pig']
Bird: parrot

Fish: salmon
```

Example 3:

```
fauna = ("parrot", "salmon", "cat", "dog", "horse", "pig")
(bird, fish, *animals) = fauna
print("Animals:", animals)
print("Bird:", bird)

print("Fish:", fish)
```

Output:

```
Animals: ['cat', 'dog', 'horse', 'pig']
Bird: parrot

Fish: salmon
```

Python Sets

Sets are unordered collection of data items. They store multiple items in a single variable. Sets items are separated by commas and enclosed within curly brackets {}. Sets are unchangeable, meaning you cannot change items of the set once created. Sets do not contain duplicate items.

Example:

```
info = {"Carla", 19, False, 5.9, 19}

print(info)
```

Output:

```
{False, 19, 5.9, 'Carla'}
```

Here we see that the items of set occur in random order and hence they cannot be accessed using index numbers. Also sets do not allow duplicate values.

Accessing set items:

Using a For loop

You can access items of set using a for loop.

Example:

```
info = {"Carla", 19, False, 5.9}
for item in info:

    print(item)
```

Output:

```
False
Carla
19

5.9
```

Add/Remove Set Items

Add items to set:

If you want to add a single item to the set use the add() method.
Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities.add("Helsinki")

print(cities)
```

Output:

```
{'Tokyo', 'Helsinki', 'Madrid', 'Berlin', 'Delhi'}
```

If you want to add more than one item, simply create another set or any other iterable object(list, tuple, dictionary), and use the update() method to add it into the existing set.
Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Helsinki", "Warsaw", "Seoul"}
cities.update(cities2)

print(cities)
```

Output:

```
{'Seoul', 'Berlin', 'Delhi', 'Tokyo', 'Warsaw', 'Helsinki', 'Madrid'}
```

Remove items from set:

We can use remove() and discard() methods to remove items form list.
Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities.remove("Tokyo")

print(cities)
```

Output:

```
{'Delhi', 'Berlin', 'Madrid'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities.discard("Delhi")

print(cities)
```

Output:

```
{'Berlin', 'Tokyo', 'Madrid'}
```

The main difference between remove and discard is that, if we try to delete an item which is not present in set, then remove() raises an error, whereas discard() does not raise any error.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities.remove("Seoul")

print(cities)
```

Output:

```
KeyError: 'Seoul'
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities.discard("Seoul")

print(cities)
```

Output:

```
{'Madrid', 'Delhi', 'Tokyo', 'Berlin'}
```

There are various other methods to remove items from the set: pop(), del(), clear().

pop():

This method removes the last item of the set but the catch is that we don't know which item gets popped as sets are unordered. However, you can access the popped item if you assign the pop() method to a variable.
Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
item = cities.pop()
print(cities)

print(item)
```

Output:

```
{'Tokyo', 'Delhi', 'Berlin'}

Madrid
```

del:

del is not a method, rather it is a keyword which deletes the set entirely.
Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
del cities

print(cities)
```

Output:

```
NameError: name 'cities' is not defined
```

We get an error because our entire set has been deleted and there is no variable called cities which contains a set.

What if we don't want to delete the entire set, we just want to delete all items within that set?

clear():

This method clears all items in the set and prints an empty set.
Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities.clear()
```

```
print(cities)
```

Output:

```
set()
```

Check if item exists

You can also check if an item exists in the set or not.

Example 1:

```
info = {"Carla", 19, False, 5.9}
if "Carla" in info:
    print("Carla is present.")
else:
    print("Carla is absent.")
```

Output:

```
Carla is present.
```

Example 2:

```
info = {"Carla", 19, False, 5.9}
if "Carmen" in info:
    print("Carmen is present.")
else:
    print("Carmen is absent.")
```

Output:

```
Carmen is absent.
```

Join Sets

Sets in python more or less work in the same way as sets in mathematics. We can perform operations like union and intersection on the sets just like in mathematics.

I. union() and update():

The union() and update() methods prints all items that are present in the two sets. The union() method returns a new set whereas update() method adds item into the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities3 = cities.union(cities2)

print(cities3)
```

Output:

```
{'Tokyo', 'Madrid', 'Kabul', 'Seoul', 'Berlin', 'Delhi'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities.update(cities2)

print(cities)
```

Output:

```
{'Berlin', 'Madrid', 'Tokyo', 'Delhi', 'Kabul', 'Seoul'}
```

II. intersection and intersection_update():

The intersection() and intersection_update() methods prints only items that are similar to both the sets. The intersection() method returns a new set whereas intersection_update() method updates into the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities3 = cities.intersection(cities2)

print(cities3)
```

Output:

```
{'Madrid', 'Tokyo'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities.intersection_update(cities2)

print(cities)
```

Output:

```
{'Tokyo', 'Madrid'}
```

III. symmetric_difference and symmetric_difference_update():

The symmetric_difference() and symmetric_difference_update() methods prints only items that are not similar to both the sets. The symmetric_difference() method returns a new set whereas symmetric_difference_update() method updates into the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
cities3 = cities.symmetric_difference(cities2)
```

Output:

```
{'Kabul', 'Delhi', 'Berlin', 'Seoul'}
```

IV. difference() and difference_update():

The difference() and difference_update() methods prints only items that are only present in the original set and not in both the sets. The difference() method returns a new set whereas difference_update() method updates into the existing set from another set.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Seoul", "Kabul", "Delhi"}
cities3 = cities.difference(cities2)

print(cities3)
```

Output:

```
{'Tokyo', 'Madrid', 'Berlin'}
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Seoul", "Kabul", "Delhi"}

print(cities.difference(cities2))
```

Output:

```
{'Tokyo', 'Berlin', 'Madrid'}
```

Set Methods

Apart from the methods we discussed earlier in the chapter there are some more methods that we can use to manipulate sets.

What if you want to check if items of a particular set are present in another set?
There are a few methods to check this.

• isdisjoint():

The isdisjoint() method checks if items of given set are present in another set. This method returns False if items are present, else it returns True.
Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
print(cities.isdisjoint(cities2))
```

Output:

```
False
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Seoul", "Kabul"}
print(cities.isdisjoint(cities2))
```

Output:

```
True
```

• issuperset():

The issuperset() method checks if all the items of a particular set are present in the original set. It returns True if all the items are present, else it returns False.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Seoul", "Kabul"}
print(cities.issuperset(cities2))
cities3 = {"Seoul", "Madrid","Kabul"}
print(cities.issuperset(cities3))
```

Output:

```
False
False
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Delhi", "Madrid"}
print(cities.issuperset(cities2))
```

Output:

```
True
```


- **issubset():**

The issubset() method checks if all the items of the original set are present in the particular set. It returns True if all the items are present, else it returns False.

Example 1:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Delhi", "Madrid"}
print(cities2.issubset(cities))
```

Output:

```
True
```

Example 2:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}
cities2 = {"Seoul", "Kabul"}
print(cities2.issubset(cities))
cities3 = {"Seoul", "Madrid", "Kabul"}
print(cities3.issubset(cities))
```

Output:

```
False
False
```

Python Dictionaries

Dictionaries are ordered collection of data items. They store multiple items in a single variable. Dictionaries items are key-value pairs that are separated by commas and enclosed within curly brackets {}.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}

print(info)
```

Output:

```
{'name': 'Karan', 'age': 19, 'eligible': True}
```


Access Items

Accessing Dictionary items:

I. Accessing single values:

Values in a dictionary can be accessed using keys. We can access dictionary values by mentioning keys either in square brackets or by using get method.
Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info['name'])
print(info.get('eligible'))
```

Output:

```
Karan
True
```

II. Accessing multiple values:

We can print all the values in the dictionary using values() method.
Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info.values())
```

Output:

```
dict_values(['Karan', 19, True])
```

III. Accessing keys:

We can print all the keys in the dictionary using keys() method.
Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info.keys())
```

Output:

```
dict_keys(['name', 'age', 'eligible'])
```

IV. Accessing key-value pairs:

We can print all the key-value pairs in the dictionary using items() method.
Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}

print(info.items())
```

Output:

```
dict_items([('name', 'Karan'), ('age', 19), ('eligible', True)])
```


Add/Remove Items

Adding items to dictionary:

There are two ways to add items to a dictionary.

I. Create a new key and assign a value to it:

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info)
info['DOB'] = 2001

print(info)
```

Output:

```
{'name': 'Karan', 'age': 19, 'eligible': True}

{'name': 'Karan', 'age': 19, 'eligible': True, 'DOB': 2001}
```

II. Use the update() method:

The update() method updates the value of the key provided to it if the item already exists in the dictionary, else it creates a new key-value pair.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info)
info.update({'age':20})
info.update({'DOB':2001})

print(info)
```

Output:

```
{'name': 'Karan', 'age': 19, 'eligible': True}

{'name': 'Karan', 'age': 20, 'eligible': True, 'DOB': 2001}
```

Removing items from dictionary:

There are a few methods that we can use to remove items from dictionary.

clear(): The clear() method removes all the items from the list.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
info.clear()

print(info)
```

Output:

```
{}
```

pop(): The pop() method removes the key-value pair whose key is passed as a parameter.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
info.pop('eligible')
print(info)
```

Output:

```
{'name': 'Karan', 'age': 19}
```

popitem(): The popitem() method removes the last key-value pair from the dictionary.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}
info.popitem()
print(info)
```

Output:

```
{'name': 'Karan', 'age': 19, 'eligible': True}
```

apart from these three methods, we can also use the del keyword to remove a dictionary item.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}
del info['age']
print(info)
```

Output:

```
{'name': 'Karan', 'eligible': True, 'DOB': 2003}
```

If key is not provided, then the del keyword will delete the dictionary entirely.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}
del info

print(info)
```

Output:

```
NameError: name 'info' is not defined
```


Copy Dictionaries

We can use the copy() method to copy the contents of one dictionary into another dictionary.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}
newDictionary = info.copy()

print(newDictionary)
```

Output:

```
{'name': 'Karan', 'age': 19, 'eligible': True, 'DOB': 2003}
```

Or we can use the dict() function to make a new dictionary with the items of original dictionary.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}
newDictionary = dict(info)

print(newDictionary)
```

Output:

```
{'name': 'Karan', 'age': 19, 'eligible': True, 'DOB': 2003}
```

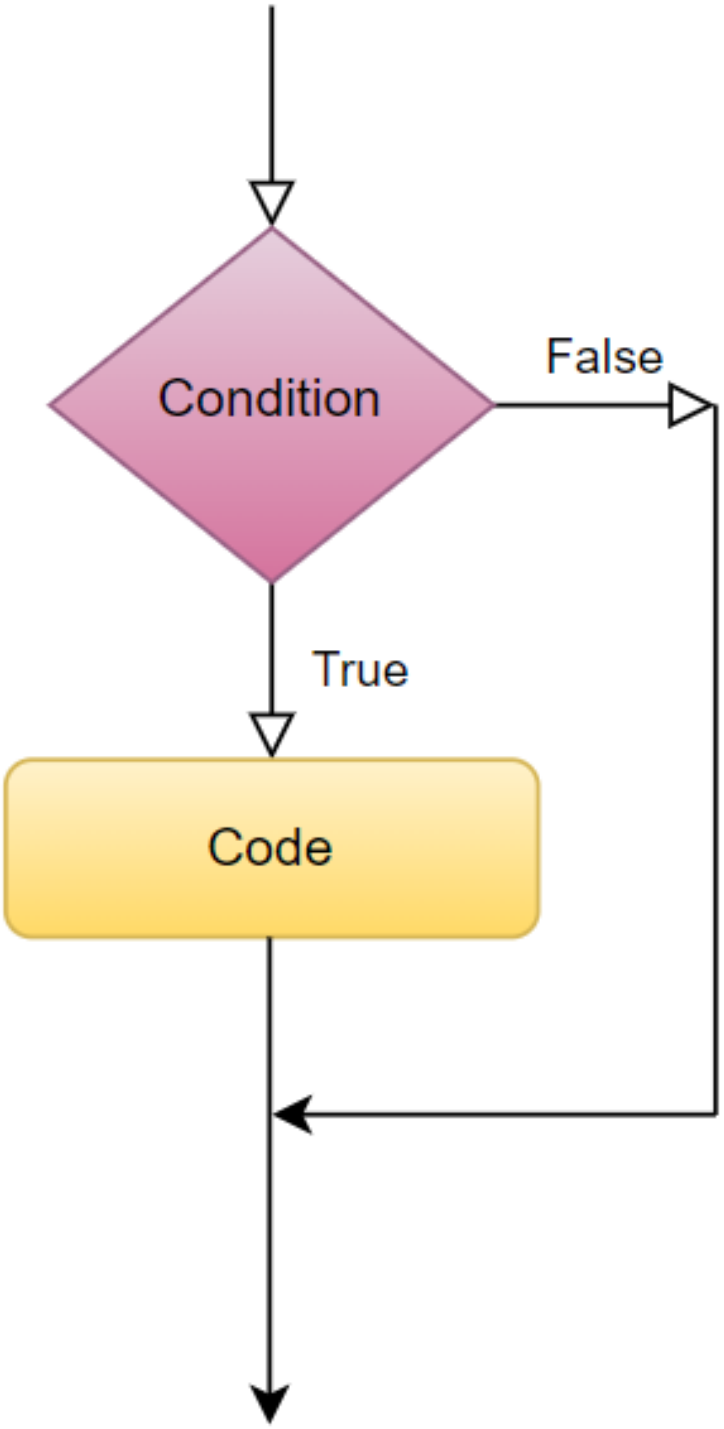

if Statement

Sometimes the programmer needs to check the evaluation of certain expression(s), whether the expression(s) evaluate to True or False. If the expression evaluates to False, then the program execution follows a different path then it would have if the expression had evaluated to True. Based on this, the conditional statements are further classified into following types; if, if……else, elif, nested if.

if Statement:

A simple if statement works on following principle,

- execute the block of code inside if statement if the expression evaluates True.
- ignore the block of code inside if statement if the expression evaluates False and return to the code outside if statement.



Example:

```
applePrice = 180
budget = 200
if (applePrice <= budget):
```

```
    print("Alexa, add 1kg Apples to the cart.")
```

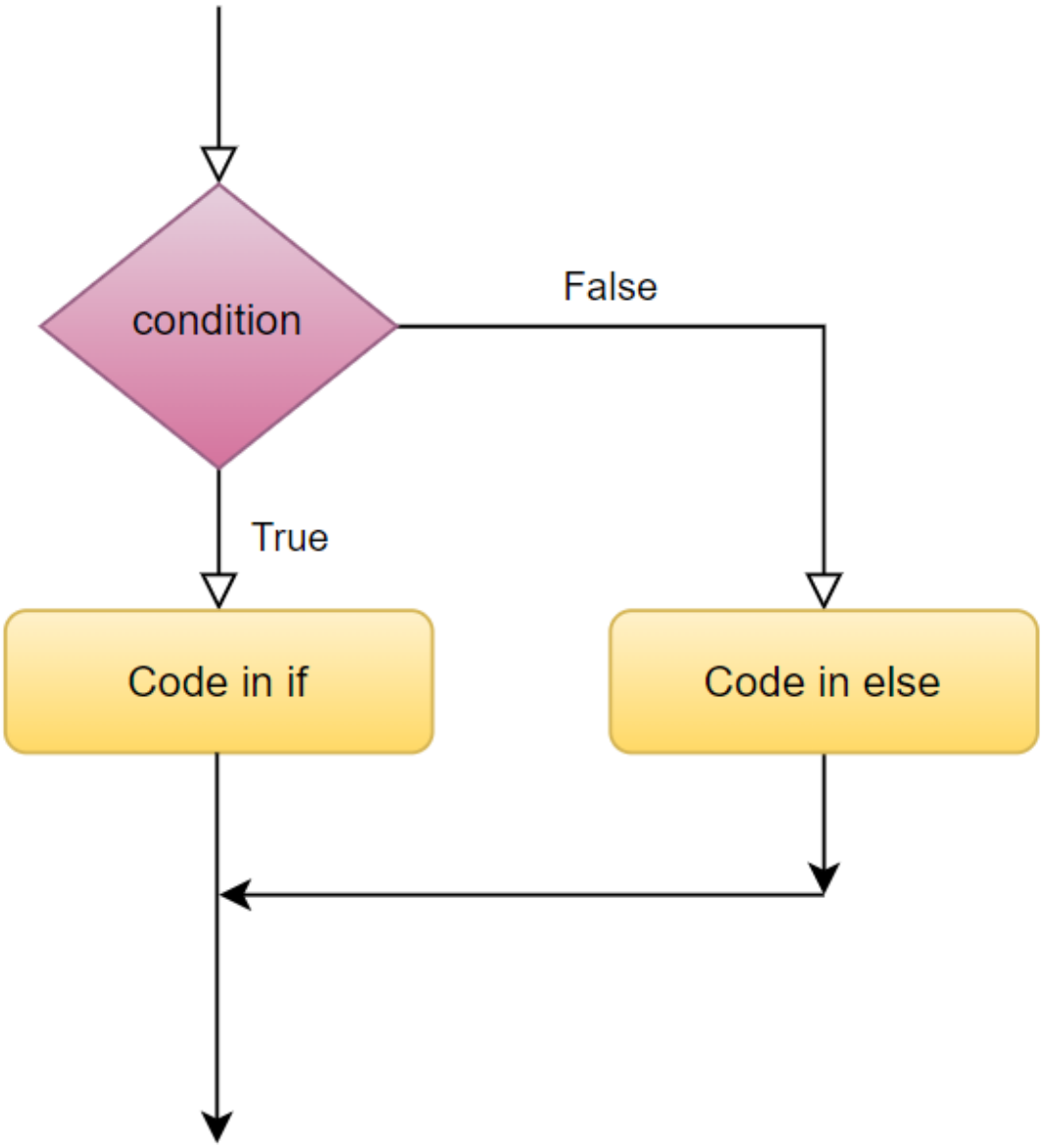
Output:

```
Alexa, add 1kg Apples to the cart.
```

if-else Statement

An if……else statement works on the following principle,

- execute the block of code inside if statement if the expression evaluates True. After execution return to the code out of the if……else block.
- execute the block of code inside else statement if the expression evaluates False. After execution return to the code out of the if……else block.



Example:

```
applePrice = 210
budget = 200
if (applePrice <= budget):
    print("Alexa, add 1kg Apples to the cart.")
else:
    print("Alexa, do not add Apples to the cart.")
```

Output:

```
Alexa, do not add Apples to the cart.
```

elif Statement

Sometimes, the programmer may want to evaluate more than one condition, this can be done using an elif statement.

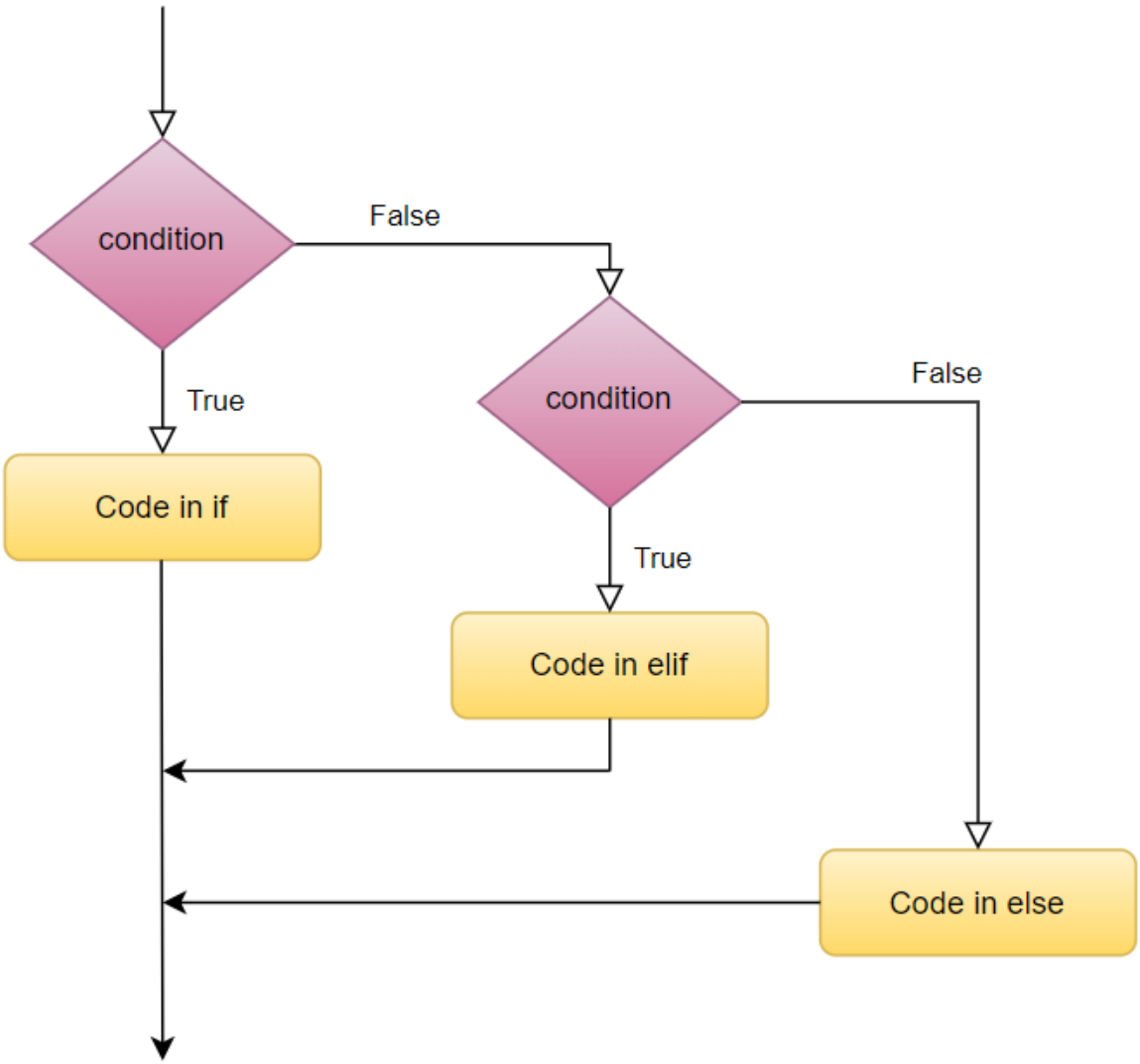
An elif statement works on the following principle,

- execute the block of code inside if statement if the initial expression evaluates to True. After execution return to the code out of the if block.
- execute the block of code inside the first elif statement if the expression inside it evaluates True. After execution return to the code out of the if block.
- execute the block of code inside the second elif statement if the expression inside it evaluates True. After execution return to the code out of the if block.

  .

  .

  .
- execute the block of code inside the nth elif statement if the expression inside it evaluates True. After execution return to the code out of the if block.
- execute the block of code inside else statement if none of the expression evaluates to True. After execution return to the code out of the if block.



Example:

```
num = 0
if (num < 0):
    print("Number is negative.")
elif (num == 0):
    print("Number is Zero.")
else:
    print("Number is positive.")
```

Output:

```
Number is Zero.
```

Nested if Statement

We can use if, if….else, elif statements inside other if statements.

Example:

```
num = 18
if (num < 0):
    print("Number is negative.")
elif (num > 0):
    if (num <= 10):
        print("Number is between 1-10")
    elif (num > 10 and num <= 20):
        print("Number is between 11-20")
    else:
        print("Number is greater than 20")
else:

    print("Number is zero")
```

Output:

```
Number is between 11-20
```

Python for Loop

Sometimes a programmer wants to execute a group of statements a certain number of times. This can be done using loops. Based on this loops are further classified into following types; for loop, while loop, nested loops.

for Loop

for loops can iterate over a sequence of iterable objects in python. Iterating over a sequence is nothing but iterating over strings, lists, tuples, sets and dictionaries.

Example: iterating over a string:

```
name = 'Abhishek'
for i in name:

    print(i, end=", ")
```

Output:

```
A, b, h, i, s, h, e, k,
```

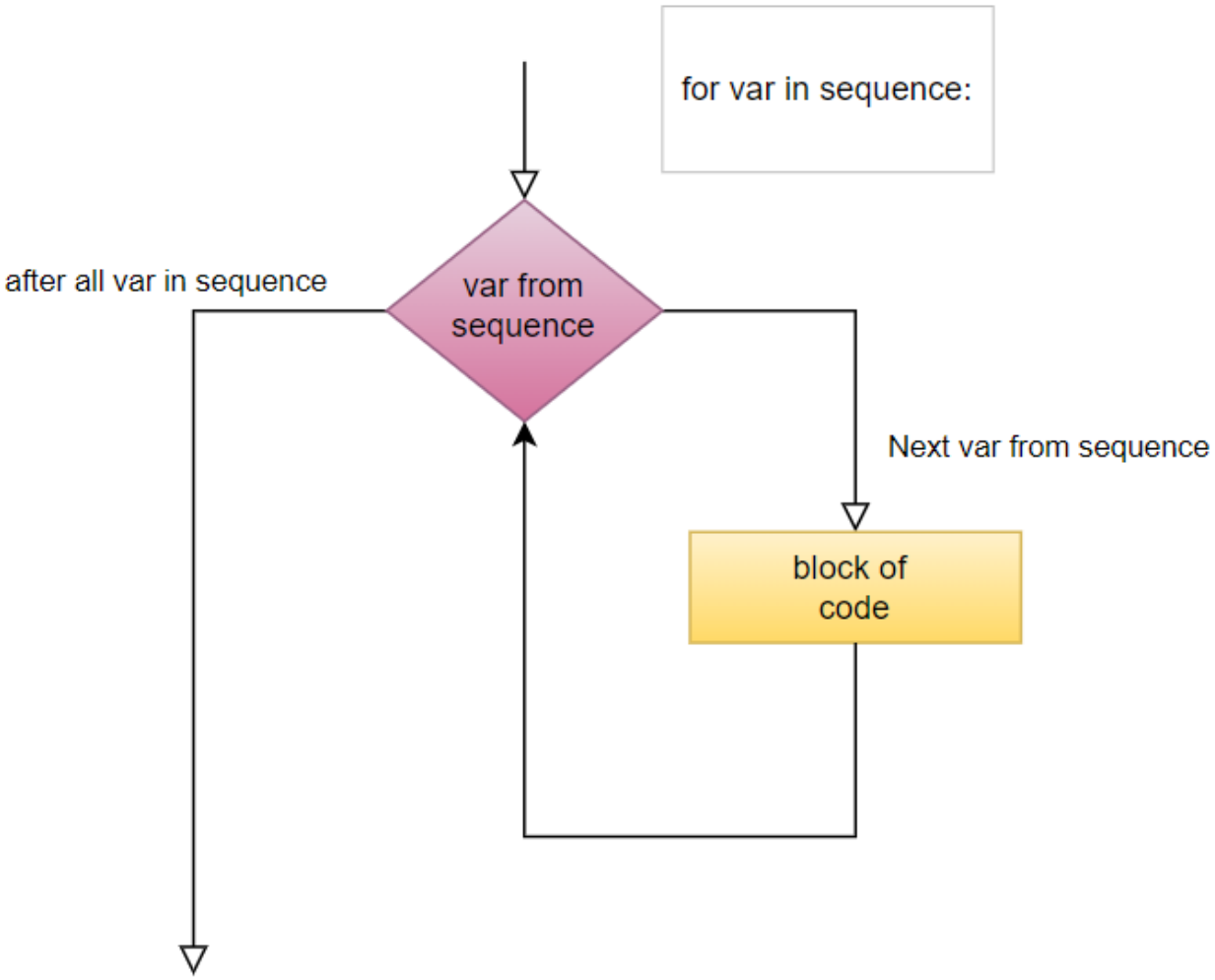
Example: iterating over a tuple:

```
colors = ("Red", "Green", "Blue", "Yellow")
for x in colors:

    print(x)
```

Output:

```
Red
Green
Blue

Yellow
```

Similarly, we can use loops for lists, sets and dictionaries.



What if we do not want to iterate over a sequence? What if we want to use for loop for a specific number of times?

Here, we can use the range() function.

Example:

```
for k in range(5):
    print(k)
```

Output:

```
0
1
2
3
4
```

Here, we can see that the loop starts from 0 by default and increments at each iteration.

But we can also loop over a specific range.
Example:

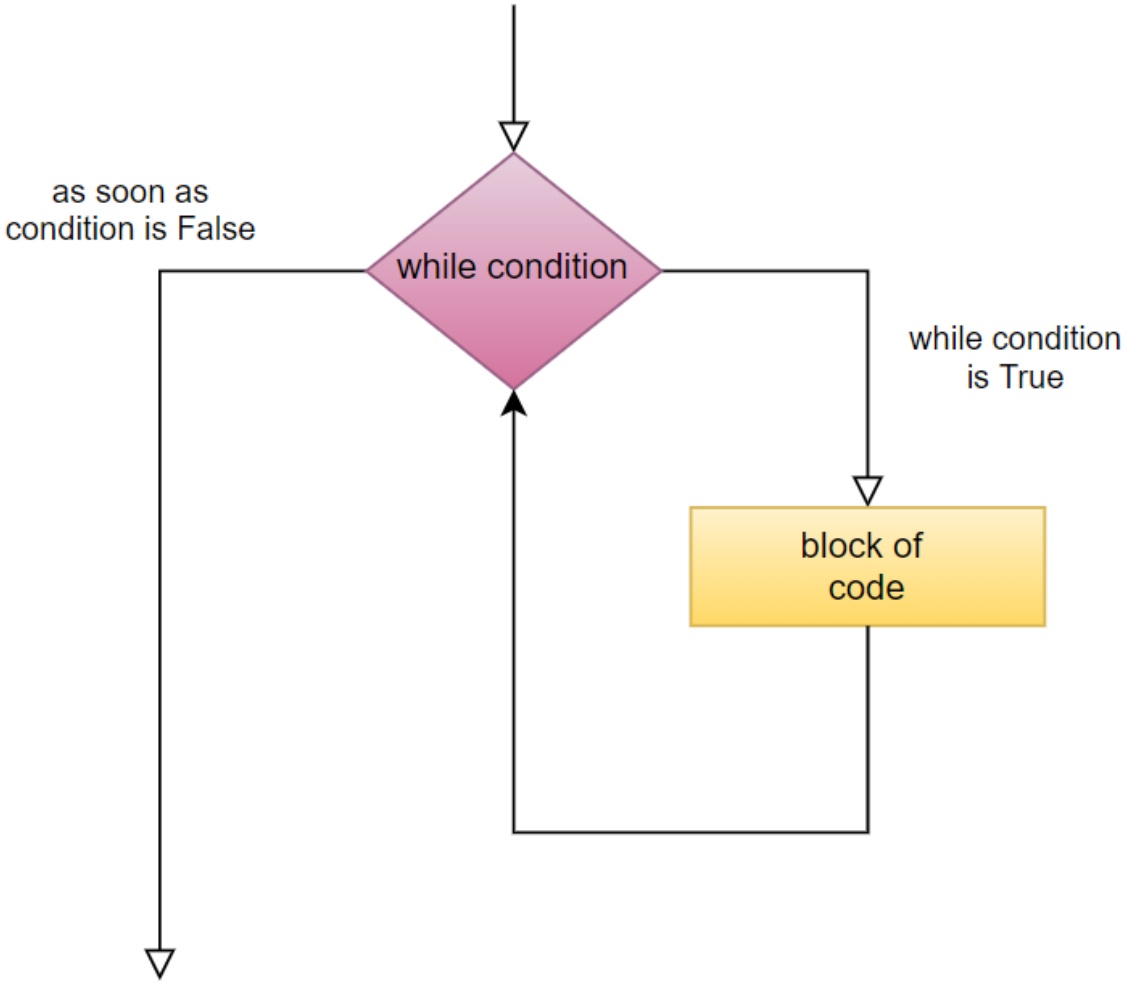
```
for k in range(4,9):
    print(k)
```

Output:

```
4
5
6
7
8
```

Python while Loop

As the name suggests, while loops execute statements while the condition is True. As soon as the condition becomes False, the interpreter comes out of the while loop.



Example:

```
count = 5
while (count > 0):
    print(count)

    count = count - 1
```

Output:

```
5
4
```

```
3
2
1
```

Here, the count variable is set to 5 which decrements after each iteration. Depending upon the while loop condition, we need to either increment or decrement the counter variable (the variable count, in our case) or the loop will continue forever.

We can even use the else statement with the while loop. Essentially what the else statement does is that as soon as the while loop condition becomes False, the interpreter comes out of the while loop and the else statement is executed.

Example:

```
x = 5
while (x > 0):
    print(x)
    x = x - 1
else:
    print('counter is 0')
```

Output:

```
5
4
3
2
1
counter is 0
```

Nested Loops

We can use loops inside other loops, such types of loops are called as nested loops.

Example: nesting for loop in while loop

```
while (i<=3):
    for k in range(1, 4):
        print(i, "*", k, "=", (i*k))
    i = i + 1

    print()
```

Output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

Example: nesting while loop in for loop

```
for i in range(1, 4):
    k = 1
    while (k<=3):
        print(i, "*", k, "=", (i*k))
        k = k + 1

    print()
```

Output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3

2 * 1 = 2
```

```
2 * 2 = 4
2 * 3 = 6

3 * 1 = 3
3 * 2 = 6

3 * 3 = 9
```

Control Statements

There are three control statements that can be used with for and while loops to alter their behaviour.
They are pass, continue and break.

1. pass:

Whenever loops, functions, if statements, classes, etc are created, it is needed that we should write a block of code in it. An empty code inside loop, if statement or function will give an error.
Example:

```
i = 1

while (i<5):
```

Output:

```
IndentationError: expected an indented block
```

To avoid such an error and to continue the code execution, pass statement is used. pass statement acts as a placeholder for future code.
Example:

```
i = 1
while (i<5):
    pass

for j in range(5):
    pass

if (i == 2):

    pass
```

The above code does not give an error.

2. continue:

This keyword is used in loops to end the current iteration and continue the next iteration of the loop. Sometimes within a loop, we might need to skip a specific iteration. This can be done using the continue keyword.

Example 1:

```
for i in range(1,10):
    if(i%2 == 0):
        continue
    print(i)
```

Output:

```
1
3
5
7
9
```

Example 2:

```
i = 1
while (i <= 10):
    i = i + 1
    if (i%2 != 0):
        continue
    print(i)
```

Output:

```
2
4
6
8
10
```

3. break:

The break keyword is used to bring the interpreter out of the loop and into the main body of the program. Whenever the break keyword is used, the

loop is terminated and the interpreter starts executing the next series of statements within the main program.

Example 1:

```
i = 1
while (i <= 10):
    i = i + 1
    if (i == 5):
        break

    print(i)
```

Output:

```
2
3

4
```

Example 2:

```
for i in range(1, 10):
    print(i)
    if (i == 5):

        break
```

Output:

```
1
2
3
4

5
```

Python Functions

A function is a block of code that performs a specific task whenever it is called. In bigger programs, where we have large amounts of code, it is advisable to create or use existing functions that make the program flow organized and neat.

There are two types of functions:

- built-in functions
- user-defined functions

1. built-in functions:

These functions are defined and pre-coded in python. Some examples of built-in functions are as follows:
min(), max(), len(), sum(), type(), range(), dict(), list(), tuple(), set(), print(), etc.

2. user-defined functions:

We can create functions to perform specific tasks as per our needs. Such functions are called user-defined functions.
Syntax:

```
def function_name(parameters):
    Code and Statements
```

- Create a function using the def keyword, followed by a function name, followed by a paranthesis (()) and a colon(:).
- Any parameters and arguments should be placed within the parentheses.
- Rules to naming function are similar to that of naming variables.
- Any statements and other code within the function should be indented.

Call a function:

We call a function by giving the function name, followed by parameters (if any) in the parenthesis.

Example:

```
def name(fname, lname):
    print("Hello,", fname, lname)

name("Sam", "Wilson")
```

Output:

```
Hello, Sam Wilson
```

Function Arguments

There are four types of arguments that we can provide in a function:

- Default Arguments
- Keyword Arguments
- Required Arguments
- Variable-length Arguments

Default arguments:

We can provide a default value while creating a function. This way the function assumes a default value even if a value is not provided in the function call for that argument.

Example:

```
def name(fname, mname = "Jhon", lname = "Whatson"):
    print("Hello,", fname, mname, lname)

name("Amy")
```

Output:

```
Hello, Amy Jhon Whatson
```

Keyword arguments:

We can provide arguments with key = value, this way the interpreter recognizes the arguments by the parameter name. Hence, the the order in which the arguments are passed does not matter.

Example:

```
def name(fname, mname, lname):
    print("Hello,", fname, mname, lname)

name(mname = "Peter", lname = "Wesker", fname = "Jade")
```

Output:

```
Hello, Jade Peter Wesker
```

Required arguments:

In case we don't pass the arguments with a key = value syntax, then it is necessary to pass the arguments in the correct positional order and the number of arguments passed should match with actual function definition.
Example 1: when number of arguments passed does not match to the actual function definition.

```
def name(fname, mname, lname):
    print("Hello,", fname, mname, lname)

name("Peter", "Quill")
```

Output:

```
name("Peter", "Quill")
TypeError: name() missing 1 required positional argument: 'lname'
```

Example 2: when number of arguments passed matches to the actual function definition.

```
def name(fname, mname, lname):
    print("Hello,", fname, mname, lname)

name("Peter", "Ego", "Quill")
```

Output:

```
Hello, Peter Ego Quill
```

Variable-length arguments:

Sometimes we may need to pass more arguments than those defined in the actual function. This can be done using variable-length arguments.

There are two ways to achieve this:

- **Arbitrary Arguments:**

While creating a function, pass a * before the parameter name while defining the function. The function accesses the arguments by processing them in the form of tuple.

Example:

```
def name(*name):
    print("Hello,", name[0], name[1], name[2])

name("James", "Buchanan", "Barnes")
```

Output:

```
Hello, James Buchanan Barnes
```

- **Keyword Arbitrary Arguments:**

While creating a function, pass a * before the parameter name while defining the function. The function accesses the arguments by processing them in the form of dictionary.

Example:

```
def name(**name):
    print("Hello,", name["fname"], name["mname"], name["lname"])

name(mname = "Buchanan", lname = "Barnes", fname = "James")
```

Output:

```
Hello, James Buchanan Barnes
```

return Statement

The return statement is used to return the value of the expression back to the main function.

Example:

```
def name(fname, mname, lname):
    return "Hello, " + fname + " " + mname + " " + lname

print(name("James", "Buchanan", "Barnes"))
```

Output:

```
Hello, James Buchanan Barnes
```

Python Recursion

We can let the function call itself, such a process is known as calling a function recursively in python.

Example:

```
def factorial(num):
    if (num == 1 or num == 0):
        return 1
    else:
        return (num * factorial(num - 1))

# Driver Code
num = 7;
print("number: ",num)
print("Factorial: ",factorial(num))
```

Output:

```
number:  7
Factorial:  5040
```

Python Modules

Python modules are python files that contain python code that we can use within our python files ensuring simplicity and code reusability.

Here are some popular python built-in modules:
csv, datetime, json, math, random, sqlite3, statistics, tkinter, turtle, etc.

Example: importing math module:

```
import math

print("Sin(0) =", math.sin(0))
print("Sin(30) =", math.sin(math.pi/6))
print("Sin(45) =", math.sin(math.pi/4))
print("Sin(60) =", math.sin(math.pi/3))

print("Sin(90) =", math.sin(math.pi/2))
```

Output:

```
Sin(0) = 0.0
Sin(30) = 0.49999999999999994
Sin(45) = 0.7071067811865476
Sin(60) = 0.8660254037844386

Sin(90) = 1.0
```

Apart from this we can create our own modules and use them within other python files.

A. Creating and using module:

Write code and save file with extension .py .

Example: creating file module.py and writing some code:

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

def sub(a, b):
    return (a-b)
```

```
def mul(a, b):
    return (a*b)

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

Now we can use this code in a different python file by simply using the import statement.
Example: creating file calc.py and importing module.py

```
import module
```

Now we can write code to call the functions from module.py in calc.py
Example:

```
import module

num1 = int(input("Enter first number:"))
num2 = int(input("Enter second number:"))

print("Addition", module.add(num1, num2))
print("Subtraction", module.sub(num1, num2))
print("Multiplication", module.mul(num1, num2))
print("Division", module.div(num1, num2))
```

Output:

```
Enter first number:6
Enter second number:2
Addition 8
Subtraction 4
Multiplication 12
Division 3.0
```

B. Using an alias:

We can also use alias while importing module. This way we do not need to write the whole name of the module while calling it.
Example:

```
import math as m

print("Square Root of 36 :", m.sqrt(36))
print("3 to the power of 4 :", m.pow(3,4))
```

Output:

```
Square Root of 36 : 6.0
3 to the power of 4 : 81.0
```

C. import from module:

eYou can also import specific parts that you need from a module instead of importing the entire module.

Example: creating file module.py and writing some code

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

def sub(a, b):
    return (a-b)

def mul(a, b):
    return (a*b)

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

Example: now we will only import add() and sub() function into our calc.py file.

```
from module import add, sub

num1 = int(input("Enter first number:"))
num2 = int(input("Enter second number:"))

print("Addition", add(num1, num2))
print("Subtraction", sub(num1, num2))
print("Multiplication", mul(num1, num2))
print("Division", div(num1, num2))
```

Output:

```
Enter first number:12
Enter second number:4
Addition 16
Subtraction 8
Traceback (most recent call last):
  File "d:\Python\Codes\calc.py", line 9, in <module>
    print("Multiplication", mul(num1, num2))

NameError: name 'mul' is not defined
```

As we can see that, we have only imported add() and sub() functions from module.py file. Hence we get output for these functions, but we get error if we use any other functions.

But what if want to import everything from a module but we do not need to prefix module name again and again?
Use an asterisk(*) while importing.

Example:

```
from module import *

num1 = int(input("Enter first number:"))
num2 = int(input("Enter second number:"))

print("Addition", add(num1, num2))
print("Subtraction", sub(num1, num2))
print("Multiplication", mul(num1, num2))

print("Division", div(num1, num2))
```

Output:

```
Enter first number:12
Enter second number:4
Addition 16
Subtraction 8
Multiplication 48

Division 3.0
```

As we can see, by using an asterisk(*), we can directly call the functions from the imported module without prefixing it with module name.

D. dir() function:

The dir() function lists all the function names (or variable names) in a module.

Example:

```
from operator import mod
import math

lst1 = dir(math)

print(lst1)
```

Output:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm',
'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Python Packages

Python packages are essentially folders that contain many python modules. As such, packages help us to import modules from different folders.

Here are some packages provided by python,
NumPy, SciPy, Pandas, Seaborn, sklearn, Matplotlib, etc.

We will learn more about these python packages in later chapters.

Python OOPS

A class is a blueprint or a template for creating objects while an object is an instance or a copy of the class with actual values.

Creating a Class:

Create a class using the class keyword.
Example:

```
class Details:
    name = "Simran"
    age = 20
```

Creating an Object:

Now create object of the class.
Example:

```
obj1 = Details()
print(obj1.name)
print(obj1.age)
```

Now we can print values:
Example:

```
class Details:
    name = "Simran"
    age = 20

obj1 = Details()
print(obj1.name)
print(obj1.age)
```

Output:

```
Simran    20
```

self method

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
It must be provided as the extra parameter inside the method definition.

Example:

```
class Details:
    name = "Simran"
    age = 20

    def desc(self):
        print("My name is", self.name, "and I'm", self.age, "years old.")

obj1 = Details()
obj1.desc()
```

Output:

```
My name is Simran and I'm 20 years old.
```

__init__ method

The __init__ method is used to initialize the object's state and contains statements that are executed at the time of object creation.

Example:

```
class Details:
    def __init__(self, animal, group):
        self.animal = animal
        self.group = group

obj1 = Details("Crab", "Crustaceans")
print(obj1.animal, "belongs to the", obj1.group, "group.")
```

Output:

```
Crab belongs to the Crustaceans group.
```

We can also modify and delete objects and their properties:

Example:

```
class Details:
    def __init__(self, animal, group):
        self.animal = animal
        self.group = group

obj1 = Details("Crab", "Crustaceans")
obj1.animal = "Shrimp"  #Modify object property
print(obj1.animal, "belongs to the", obj1.group, "group.")
```

Output:

```
Shrimp belongs to the Crustaceans group.
```

Example:

```
class Details:
    def __init__(self, animal, group):
        self.animal = animal
        self.group = group
```

```
obj1 = Details("Crab", "Crustaceans")
del obj1     #delete object entirely

print(obj1.animal, "belongs to the", obj1.group, "group.")
```

Output:

```
Traceback (most recent call last):
  File "d:\Python \Codes\class.py", line 12, in <module>
    print(obj1.animal, "belongs to the", obj1.group, "group.")

NameError: name 'obj1' is not defined
```

Python Iterators

Iterators in python are used to iterate over iterable objects or container datatypes like lists, tuples, dictionaries, sets, etc.

It consists of __iter__() and __next__() methods.
__iter__() : to initialize an iterator, use the __iter__() method.
__next__() : This method returns the next item of the sequence.

Using inbuilt iterators:

```
string = 'Hello World'
iterObj = iter(string)

while True:
    try:
        char1 = next(iterObj)
        print(char1)
    except StopIteration:
        break
```

Output:

```
H
e
l
l
o

W
o
r
l
d
```

Creating Custom iterators:

```
class multipleOf4:
  def __iter__(self):
    self.count = 0
    return self

  def __next__(self):
    if self.count <= 24:
      x = self.count
      self.count += 4
      return x
    else:
      raise StopIteration

obj1 = multipleOf4()
number = iter(obj1)

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

Output:

```
0
4
8
12
16
20
24
```

In both the above examples, we can see that there is an StopIteration statement inside the except block. This is to prevent the iteration from continuing forever.

JSON

JSON stands for JavaScript Object Notation. It is a built-in package provided in python that is used to store and exchange data.

A. Converting JSON string to python:

Example:

```
import json

# JSON String:
colors =  '["Red", "Yellow", "Green", "Blue"]'

# JSON string to python dictionary:
lst1 = json.loads(colors)

print(lst1)
```

Output:

```
['Red', 'Yellow', 'Green', 'Blue']
```

B. Converting python to JSON string:

Example:

```
import json

# python dictionary
lst1 = ['Red', 'Yellow', 'Green', 'Blue']

# Convert Python dict to JSON
jsonObj = json.dumps(lst1)

print(jsonObj)
```

Output:

```
["Red", "Yellow", "Green", "Blue"]
```

C. Conversion type:

Whenever python objects are converted to JSON, object type is changed to match that of JSON.

Example:

```
import json

print(json.dumps(22))                    #integer
print(json.dumps(6.022))                 #float
print(json.dumps("Hello World"))         #string
print(json.dumps(True))                  #True
print(json.dumps(False))                 #False

print(json.dumps(None))                  #None
```

Output:

```
22
6.022
"Hello World"
true
false

null
```

Python try...except

try….. except blocks are used in python to handle errors and exceptions. The code in try block runs when there is no error. If the try block catches the error, then the except block is executed.

Example:

```
try:
    num = int(input("Enter an integer: "))
except ValueError:
    print("Number entered is not an integer.")
```

Output 1:

```
Enter an integer: 6.022
Number entered is not an integer.
```

Output 2:

```
Enter an integer: -8.5
Number entered is not an integer.
```

Output 3:

```
Enter an integer: -36
```


In addition to try and except blocks, we also have else and finally blocks. The code in else block is executed when there is no error in try block.

Example:

```
try:
    num = int(input("Enter an integer: "))
except ValueError:
    print("Number entered is not an integer.")
else:
    print("Integer Accepted.")
```

Output 1:

```
Enter an integer: 69

Integer Accepted.
```

Output 2:

```
Enter an integer: -2.3

Number entered is not an integer.
```

The finally block is execute whatever is the outcome of try……except…..else blocks.

Example:

```
try:
    num = int(input("Enter an integer: "))
except ValueError:
    print("Number entered is not an integer.")
else:
    print("Integer Accepted.")
finally:

    print("This block is always executed.")
```

Output 1:

```
Enter an integer: 19
Integer Accepted.

This block is always executed.
```

Output 2:

```
Enter an integer: 3.142
Number entered is not an integer.

This block is always executed.
```

Python PIP

pip stands for Package Installer for Python. It is used to install and manage software packages in python that are not the part of standard python library.
In the later versions of python (3.4 and after), the pip command is pre-installed.

To Check if pip is installed in your system, type the following in command prompt:

- pip --version

```
pip 22.2 from
C:\users\yourName\appdata\local\programs\python\python39\lib\site-pac
kages\pip (python 3.9)
```

If your system does not have pip installed, you can easily download it from their official website: https://pypi.org/project/pip/
Now that pip has been installed in our system, we can download packages in system.
Example:

```
PS D: \Python\Codes> pip install numpy
Collecting numpy
  Downloading numpy-1.23.1-cp39-cp39-win_amd64.whl (14.7 MB)
     ---------------------------------------- 14.7/14.7 MB 2.9 MB/s eta 0:00:00
Installing collected packages: numpy
```

If the package already exists then the following message is displayed:

```
PS D:\Python\Codes> pip install sklearn
Requirement already satisfied: sklearn in
c:\users\yourName\appdata\local\programs\python\python39\lib\site-pac
kages (0.0)
```

To list all the installed packages:

```
PS D:\Python\Codes> pip list
Package                           Version
--------------------------------- -------------------
Flask                             2.0.3
ipython                           8.0.1
jupyter                           1.0.0
keras                             2.8.0
Kivy                              2.0.0
matplotlib                        3.4.3
mysql-connector                   2.2.9
numpy                             1.19.5
pandas                            1.4.1
pip                               22.2
plotly                            5.6.0
pygame                            2.0.0
scipy                             1.7.1
seaborn                           0.11.2
selenium                          3.141.0
sklearn                           0.0
sympy                             1.10
tensorflow                        2.8.0
```

Data & Time

Dates and timings are very important for computers. Computers keeps tracks of date and time for files, documents and various other operations. Similarly python uses date and time to perform several operations.

Example:

```
import time

print(time.time())
```

Output:

```
1658659152.7263992
```

So, what can we understand from the about output? What does a bunch of random numbers mean?
These numbers are nothing but the number of ticks since the start of January 1, 1970 also called as epoch.
Now, one might ask what are ticks?
Ticks are floating point numbers measured in units of seconds for time interval.

We can also pass the epoch inside the ctime() function (or simply use the ctime() function) to print the current time in human readable format.
Example:

```
import time

print(time.ctime())
```

Output:

```
Sun Jul 24 16:24:50 2022
```

time.sleep():

the sleep() function inside time module is used to delay execution of current time thread by the given number of seconds.

Example:

```
import time

time.sleep(10)

print(time.ctime())
```

Output:

```
Sun Jul 24 16:24:50 2022
```

As we can see, the sleep() will delay the execution 10 seconds.

Time.struct_time_Class:

Index	Field	Attribute	Value	Meaning	Format Codes
0	Year	tm_year	0000, ……., 9999	Four digit year	%Y
1	Month	tm_mon	1 – January, 2 – February, ……., 12 - December	Months in a year	%m
2	Day	tm_mday	1, 2, 3, ……., 29, 31	Days in a month	%d
3	Hour	tm_hour	0, 1, 2, ……., 22, 23	Hours in a day	%H
4	Minute	tm_min	0, 1, 2, ……., 58, 59	Minutes in an hour	%M
5	Second	tm_sec	0, 1, 2, ……., 60, 61	Seconds in a minute	%S

6	Day of Week	tm_wday	0 – Monday, 1 – Tuesday, ……., 6 – Sunday	Days in a week	%w
7	Day of Year	tm_yday	1, 2, 3, ……., 355, 356	Days in a year	%j
8	Daylight savings	tm_isdst	-1, 0, 1		

time.localtime():

To get the output of local time in struct_time format, use the localtime() function.
Example:

```
import time

print(time.localtime(1658672956.8853111))
```

Output:

```
time.struct_time(tm_year=2022, tm_mon=7, tm_mday=24, tm_hour=19,
tm_min=59, tm_sec=16, tm_wday=6, tm_yday=205, tm_isdst=0)
```

time.gmtime():

To get the output of Coordinated Universal Time in struct_time format, use the gmtime() function.
Example:

```
import time

print(time.gmtime(1658672956.8853111))
```

Output:

```
time.struct_time(tm_year=2022, tm_mon=7, tm_mday=24, tm_hour=14,
tm_min=29, tm_sec=16, tm_wday=6, tm_yday=205, tm_isdst=0)
```

time.mktime():

We also have the reverse of localtime() function that prints seconds passed since epoch in local time.

Example:

```
import time

local_time = (2022, 7, 24, 20, 14, 39, 6, 205, 0)
print(time.mktime(local_time))
```

Output:

```
1658673879.0
```

time.asctime():

The asctime() function takes struct_time and prints a single string representing it.

Example:

```
import time

local_time = (2022, 7, 24, 20, 14, 39, 6, 205, 0)
print(time.asctime(local_time))
```

Output:

```
Sun Jul 24 20:14:39 2022
```

time.strptime():

This function parses string based on given format and returns it in struct_time format.

Example:

```
import time

local_time = "24 July, 2022"

print(time.strptime(local_time, "%d %B, %Y"))
```

Output:

```
time.struct_time(tm_year=2018, tm_mon=7, tm_mday=24, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=1, tm_yday=205, tm_isdst=-1)
```

Calendar:

We can even print a calendar for a particular month. This can be done using the calendar module.

Example:

```
import calendar

print(calendar.month(2022, 7))
```

Output:

```
    July 2022
Mo Tu We Th Fr Sa Su
                   1   2   3
 4    5    6    7   8   9  10
11  12  13 14  15 16 17
18  19  20 21  22 23 24
25  26  27 28  29 30 31
```

Python File Handling

File handling in python lets you open, read, write, append, modify files.
Note: For the sake of this tutorial, we will assume that your text file and python code are present in the same directory.
Before we perform any operation on file, we need to open the file. This is done using the open() function.

Example: lets say we have a text file (someText.txt) with some content in it. The open() function creates a file object with read() method for reading the content.

```
file = open("someText.txt")
print(file.read())
```

Output:

```
lorem ipsum

In publishing and graphic design, Lorem ipsum is a placeholder text commonly used to demonstrate the visual form of a document or a typeface without relying on meaningful content.
```

There are various modes in which we can open files.

- read (r): This mode opens the file for reading only and gives an error if the file does not exist. This is the default mode if no mode is passed as a parameter.
- write (w): This mode opens the file for writing only and creates a new file if the file does not exist.
- append (a): This mode opens the file for appending only and creates a new file if the file does not exist.
- create (x): This mode creates a file and gives an error if the file already exists.

Apart from these modes we also need to specify how the file must be handled:

- text (t): Used to handle text files.

- binary (b): used to handle binary files (images).

Read/Write Files

A. Create a File:

Creating a file is primarily done using the create (x) mode.
Example:

```
file = open("Text.txt", "x")
```

Output:

```
A file named Text.txt is created with no content.
```

B. Write onto a File:

This method writes content onto a file.
Example:

```
file = open("Text.txt", "w")
file.write("This is an example of file creation.")

file.close
```

Output:

```
This is an example of file creation.
```

If the file already exists with some content of its own, then this mode overwrites it.
Example:

```
file = open("Text.txt", "w")
file.write("This is overwritten text.")

file.close
```

Output:

```
This is overwritten text.
```

C. Read a File:

This method allows us to read the contents of the file.
Example:

```
file = open("Text2.txt", "r")
print(file.read())

file.close
```

Output:

```
Hello, I'm a Potato.
```

D. Append a File:

This method appends content into a file.
Example:

```
file = open("newText.txt", "a")
file.write("This is an example of file appending.")

file.close
```

Output:

```
This is an example of file appending.
```

However, If the file exists already with some content in it, then the new content gets appended.
Example:

```
file = open("newText.txt", "a")
file.write(" Hello, I'm appending")

file.close
```

Output:

```
This is an example of file appending. Hello, I'm appending
```

Contact Us

🏠 Nearby CCRC College Balkumari Bridge,
Koteshwor, Kathmandu Nepal
WhatsApp / Phone Call
📱 9851344071 / 9818968546
📱Phone Call:
9806393939 / 9860267997
📩 infotech@sipalaya.com
🌐 https://sipalaya.com/
🌐Google Map Link:
https://goo.gl/maps/HY8oiYmEUzQ2nVGRA

Happy Coding !!
Thank You !!