Python Course

Python Basics

What is Python?

- Python is a popular and High Level programming language.
- It was created by Guido van Rossum, and released in 1991.

Why to Learn Python?

- Python has a simple syntax similar to the English language.
- Python is Open Source which means its available free of cost.
- Python is simple and so easy to learn
- Python has powerful development libraries include AI, ML etc.
- Python is much in demand and ensures high salary
- Python can be used on a server to create web applications.
- Python works on different platforms (Windows, Mac, Linux).

Careers with Python

- Python developer
- Artificial Intelligence

- Game developer
- Full-stack developer
- Machine learning engineer
- Data scientist
- Data analyst
- Data engineer
- Many more other roles

Applications on Python

- Netflix
- BGMI
- Robots
- many more

Features of Python

- Easy-to-learn
- Easy-to-read
- Easy-to-maintain
- Easy to Connect with Databases
- Open Source
- Large Standard libraries like Numpy,Pandas, Matplotlib etc.....

Python Install

By Clicking below link you can get official Python Website

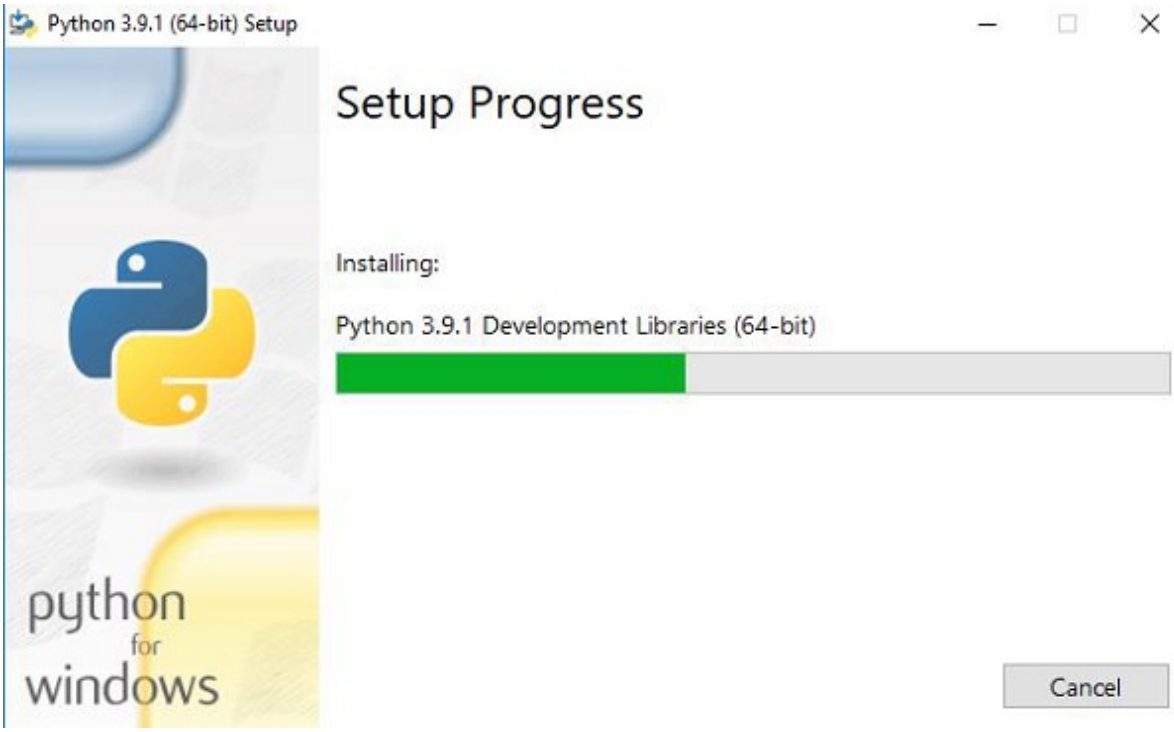
link: https://www.python.org/

Steps to install Python

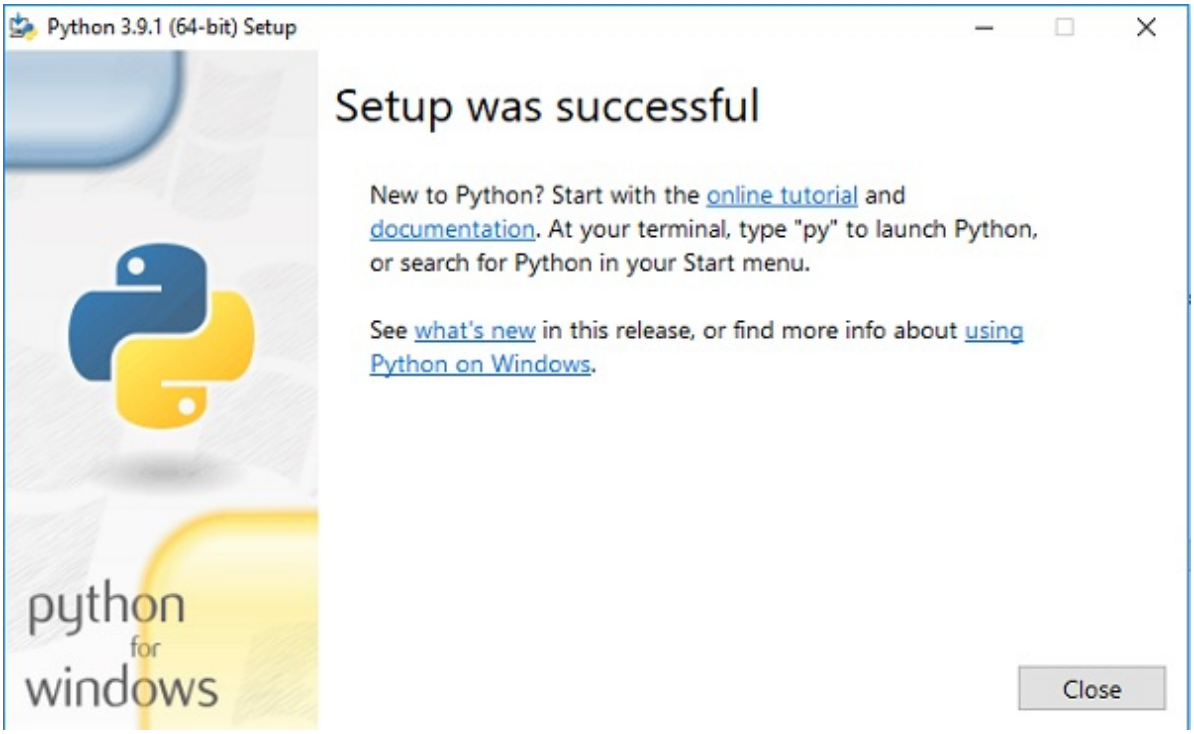
- Go to Python.org website
- Click on Download
- Download Requried Python Version
- After that click on file Explorer
- Click on Download Folder
- Double Tab on Python Installer

In [ ]:

- On clicking the Install Now, The installation process starts.



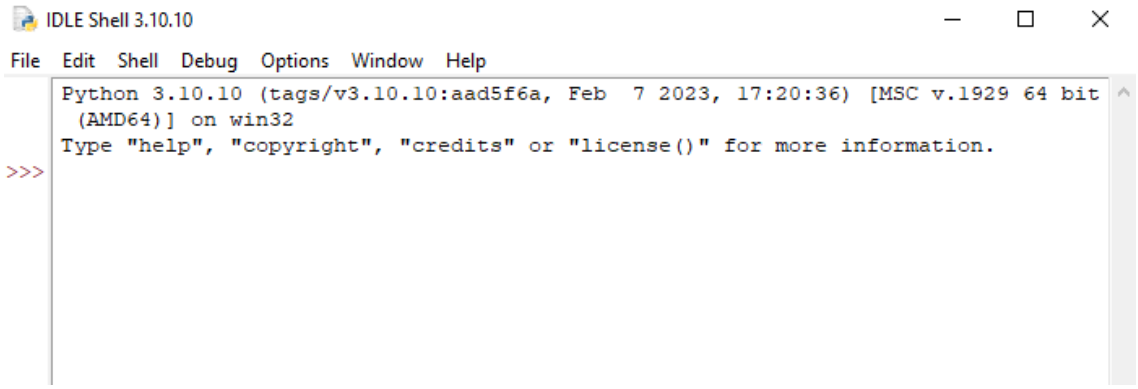
- The installation process will take few minutes to complete and once the installation is successful, the following screen is displayed.



- Verify the Python version -- Open command prompt -- Type python --version

```
C:\Users\ashub>python --version
Python 3.10.11
```

- Finaly you get below interface after installization



In [ ]:

Python First Program

In [1]:
```
print("Hello World")
```

```
Hello World
```

In [1]:
```
import sys

# Print only the Python version as a string
print("Python version:", sys.version)
```

```
Python version: 3.11.5 | packaged by Anaconda, Inc. | (main, Sep 11 2023, 13:26:23)
[MSC v.1916 64 bit (AMD64)]
```

Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Python uses indentation to indicate a block of code.

In [4]:
```
#with Indentation
if 10 > 2:
    print("Ten is greater than two!")
```

```
Ten is greater than two!
```

In [4]:
```
# Without Indentation
if 5 > 2:
print("Five is greater than two!")
```

```
  Cell In[4], line 3
    print("Five is greater than two!")
    ^
IndentationError: expected an indented block after 'if' statement on line 2
```

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments starts with a #, and Python will ignore them

In [9]:
```
#hfkasfhhkfasjhfkjahfianfiasifhaikjf
```

In [4]:
```
#This is a comment
print("Hello, World!")
```

```
Hello, World!
```

In [ ]:
```
print("Hello, World!") #This is a comment
```

Multiline Comments

- Python does not really have a syntax for multiline comments.
- To add a multiline comment you could insert a # for each line:

In [5]:
```
#This is a comment
#written in
```

```
#more than just one line
print("Hello, World!")
```

```
Hello, World!
```

Python Identifiers

- A Python identifier is a name used to identify a variable, function, class, module or other object.
- An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python is a case sensitive programming language. Thus, 'Manpower' and 'manpower' are two different identifiers in Python.

Python keywords

- python keywords are Reserved Words
- you cannot use them as constant or variable or any other identifier names.
- All the Python keywords contain lowercase letters

In [6]:
```
import keyword

# Get the list of Python keywords
python_keywords = keyword.kwlist

# Display the list of keywords
print(python_keywords)

len(python_keywords)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'globa
l', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise',
'return', 'try', 'while', 'with', 'yield']
```

Out[6]:
```
35
```

In [ ]:
```

```

Python Variables

- Python variables are the reserved memory locations used to store values with in a Python Program.
- This means that when you create a variable you reserve some space in the memory.
- Variables are containers for storing data values.

In [3]:
```
# creating a Variable

a = 10
print(a)
# where 'a' is variable and it is store a Value '10' by assign in it
```

```
10
```

Rules to Create a Variables

- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- A variable name cannot start with a number or any special character like $, (, * % etc.
- Python reserved keywords cannot be used naming the variable.
- Python variable names are case-sensitive which means Name and NAME are two different variables in Python.
- Variable name can't start with Digits

In [5]:
```
#vaild Variables

var = 13
var_3 = 'Python'
_var = 'Welcome to Data Science Course'
```

In [9]:
```
#invaild Variables

1_var = 565
@var = 'Data'
for = 'Python'
```

```
  Cell In[9], line 3
    1_var = 565
     ^
SyntaxError: invalid decimal literal
```

In [22]:
```
#You can get the data type of a variable with the type() function.
x = 57
y = "Data science"
print(type(x))
print(type(y))
```

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

Multiple Variables

In [4]:
```
x, y, z =  "Cherry", "Apple", "Banana"

print(x)
print(y)
print(z)
```

```
Cherry
Apple
Banana
```

In [24]:
```
#One Value to Multiple Variables
x=y=z = 100
print(x)
print(y)
print(z)
```

```
100
100
100
```

Unpack a Collection

- If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.

In [5]:
```
fruits = ["apple", "banana", "cherry"] #packed collection
x, y, z = fruits
print(x)
print(y)
print(z)
```

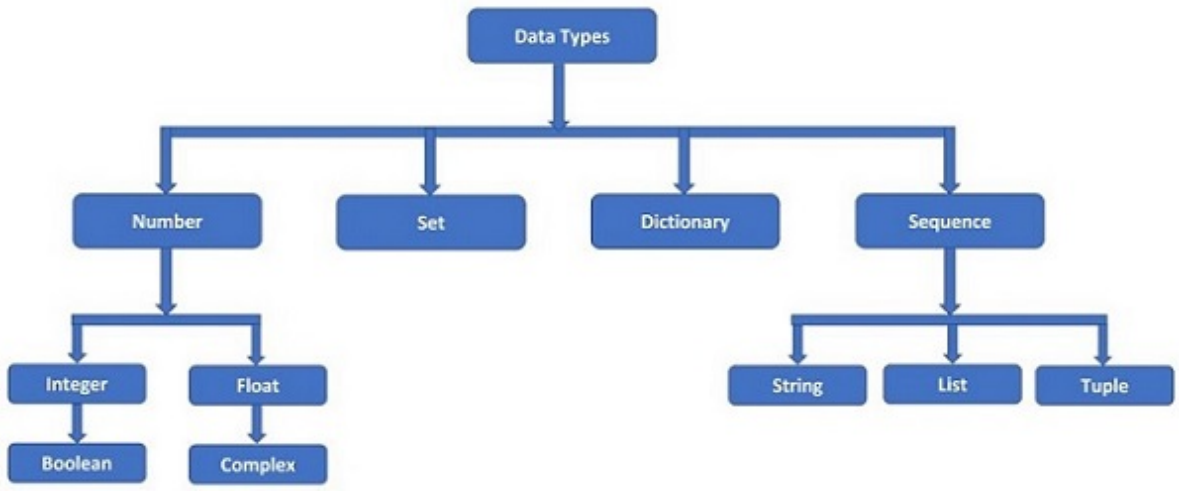
```
apple
banana
cherry
```

Python Data Types

- It defines what type of data we are going to store in a variable.
- The data stored in memory can be of many types.
- Python has various built-in data types

-- Data Type --> Examples

- Numeric --> int, float, complex
- String --> str
- Sequence --> list, tuple, range
- Mapping --> dict
- Boolean --> bool
- Set --> set



Python Numbers

There are three numeric types in Python:

- int
- float
- complex

In [1]:
```
x = 1    # int
y = 2.8  # float
z = 1j  #1+6j # complex
```

In [2]:
```
print(type(x))
print(type(y))
print(type(z))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

Type Conversion

- You can convert from one type to another with the int(), float(), and complex() methods
- Note: You cannot convert complex numbers into another number type.

In [3]:
```
x = 1    # int
y = 2.844 # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(y)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

```
1.0
2
(2.844+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

In [6]:
```
y = 2.844j
c = float(y)
c
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 2
      1 y = 2.844j
----> 2 c = float(y)
      3 c

TypeError: float() argument must be a string or a real number, not 'complex'
```

Python Booleans

- Booleans represent one of two values: True or False.

In [7]:
```
print(15 > 9)
print(13 == 9)
print(1 < 9)
```

```
True
False
True
```

Evaluate Values and Variables

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

In [9]:
```
print(bool("Hello"))
print(bool(5))
```

```
True
True
```

Python Operators

Operators are used to perform operations on variables and values. Python divides the operators in the following groups: - Arithmetic operators - Assignment operators - Comparison operators - Logical operators - Identity operators - Membership operators - Bitwise operators

Python Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations

In [11]:
```
a = 67
b = 86
```

In [12]:
```
# Addition
c = a + b
```

```
print(c)
```

```
153
```

In [13]:
```
#Subtraction
d = a - b
print(d)
```

```
-19
```

In [14]:
```
#Multiplication
f = a * b
print(f)
```

```
5762
```

In [15]:
```
#Division
g = a / b               #returns the quotient
print(g)
```

```
0.7790697674418605
```

In [16]:
```
#Modulus
h = a % b
print(h)       # It returns the remainder
```

```
67
```

In [17]:
```
#Exponentiation
j = a ** b
print(j)
```

```
1102638195149451954453861843245588305785744692829734859976572473035125802526043464573
7117862538664702513136890902785802597870420145626388338001806739753231769
```

In [18]:
```
#Floor division
k = a // b
print(k)
```

```
0
```

Assignment Operators

-- Assignment operators are used to assign values to variables.

Operator	Name
`=`	Assignment Operator
`+=`	Addition Assignment
`-=`	Subtraction Assignment
`*=`	Multiplication Assignment
`/=`	Division Assignment
`%=`	Remainder Assignment
`**=`	Exponent Assignment

In [19]:
```
# assign 10 to a
a = 10

# assign 5 to b
b = 5

# assign the sum of a and b to a
a += b      # a = a + b



print(a)
```

```
15
```

Comparison Operators

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

Operator	Meaning
==	Is Equal To
!=	Not Equal To
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

In [20]:
```
a = 50

b = 28

# equal to operator
print('a == b =', a == b)

# not equal to operator
print('a != b =', a != b)

# greater than operator
print('a > b =', a > b)

# less than operator
print('a < b =', a < b)

# greater than or equal to operator
print('a >= b =', a >= b)

# less than or equal to operator
print('a <= b =', a <= b)
```

```
a == b = False
a != b = True
a > b = True
a < b = False
a >= b = True
a <= b = False
```

Logical Operators

-- Logical operators are used to check whether an expression is True or False. -- Python logical operators are used to combile two or more conditions and check the final result.

OPERATOR	DESCRIPTION	SYNTAX
and	Returns True if both the operands are true	x and y
or	Returns True if either of the operands is true	x or y
not	Returns True if the operand is false	not x

In [21]:
```
# logical and operator
#x = 5

print(5 > 3 and 5 < 10)
```

```
True
```

In [22]:
```
# logical or operator
x = 5

print(x > 9 or x < 4)
```

```
False
```

In [23]:
```
#not operation
x = 5

print(not(x > 3 and x < 10))
```

```
False
```

Identity Operators

-- Identity whether the value is present or not

Operator	Meaning
`is`	`True` if the operands are identical (refer to the same object)
`is not`	`True` if the operands are not identical (do not refer to the same object)

In [2]:
```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'


print(x1 is not y1)  # prints False

print(x2 is y2)  # prints True


# False, as x and y are different objects in memory
```

```
False
True
```

In [27]:
```
t = [1,2,3]
i = [1,2,3]

print(t is i)
```

```
False
```

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

Membership operators

-- They are used to test whether a value or variable is found in a sequence or not

Operator	Meaning
`in`	`True` if value/variable is **found** in the sequence
`not in`	`True` if value/variable is **not found** in the sequence

In [3]:
```
x = 'Hello world'
y = {1:'a', 2:'b'}
```

```
# check if 'H' is present in x string
print('H' in x)  # prints True

# check if 'hello' is present in x string
print('hello' not in x)  # prints True

# check if '1' key is present in y
print(1 in y)  # prints True

# check if 'a' key is present in y
print('a' in y)  # prints False
```

```
True
True
True
False
```

Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name. For example, 2 is 10 in binary and 7 is 111

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

In [4]:
```
"""
The & operator compares each bit and set it to 1 if both are 1, otherwise it is set to

6 = 0110
3 = 0011
-----------
2 = 0010
===========
"""
print(6 & 3)
```

```
2
```

In [ ]:
```
"""
The | operator compares each bit and set it to 1 if one or both is 1, otherwise it is

6 = 0110
3 = 0011
---------
7 = 0111
=========

"""

print(6 | 3)
```

In [ ]:
```
"""
The ^ operator compares each bit and set it to 1 if only one is 1, otherwise (if both

6 = 0110
3 = 0011
----------
5 = 0101
==========


"""

print(6 ^ 3)
```

In [ ]:
```
"""
The ~ operator inverts each bit (0 becomes 1 and 1 becomes 0).

Inverted 3 becomes -4:
 3 = 0000000000000011
-4 = 1111111111111100

simple formal

-(n+1)
"""

print(~3)
print(~83)
```

In [5]:
```
"""
The << operator inserts the specified number of 0's (in this case 2) from the right an

If you push 00 in from the left:
 3 = 0000000000000011 # 8 4 2 1
becomes
12 = 0000000000001100

"""
print(4 << 2)
```

16

In [6]:
```
"""
The >> operator moves each bit the specified number of times to the right. Empty holes
```

```
If you move each bit 2 times to the right, 8 becomes 2:
 8 = 1000
becomes
 2 = 0010


"""

print(8 >> 2)
```

```
2
```

Precedence

-- Operator precedence describes the order in which operations are performed.

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x` `-x` `~x`	Unary plus, unary minus, and bitwise NOT
`*` `/` `//` `%`	Multiplication, division, floor division, and modulus
`+` `-`	Addition and subtraction
`<<` `>>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==` `!=` `>` `>=` `<` `<=` `is` `is not` `in` `not in`	Comparisons, identity, and membership operators
`not`	Logical NOT
`and`	AND
`or`	OR

In [31]:
```
print((6 + 3) - (6 + 3))
```

```
0
```

In [2]:
```
6+(4+7)-5
```

Out[2]:
```
12
```

In [ ]:
```
6 +11 - 5
```

In [1]:
```
17-5
```

Out[1]:
```
12
```

In [32]:
```
9+7-(4-2)*5
```

Out[32]:
```
6
```

In [ ]:
```
90-67+4*3/5
```

In [ ]:
```
8**3+(6-3)
```

Python List

-> It is commonly used data type that represents an ordered, mutable (modifiable), and iterable sequence of elements. -> Lists are used to store collections of items, and these items can be of any data type, including numbers, strings, or even other lists. -> Lists are defined using square brackets [] and elements are separated by commas.

Creation of Lists:

In [3]:
```
# Creating an empty list
my_list = []

# Creating a list with elements
numbers = [1, 2, 3, 4, 5]
names = ["Apple", "Banana", "Cherry"]
mixed_types = [1, "apple", 3.14, True]
```

In [4]:
```
print(type(mixed_types))
```

```
<class 'list'>
```

Access List Elements

In Python, lists are ordered and each item in a list is associated with a number. The number is known as a list index.

The index of the first element is 0, second element is 1 and so on.

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

In [5]:
```
languages = ["Python", "Swift", "C++"]

# access item at index 0
print(languages[0])   # Python

# access item at index 2
print(languages[2])   # C++
```

```
Python
C++
```

In [6]:
```
# pos Index =   0 1 2 4
lst1 = [1,2,3,5]
#neg index == -4 -3 -2 -1

print(lst1[-2])
```

```
3
```

In [7]:
```
languages = ["Python", "Swift", "C++", "java"]

# access item at index 0
print(languages[-1])   # java

# access item at index 2
print(languages[-2])   #c++
```

```
java
C++
```

Slicing of a List

In Python, it is possible to access a portion of a list using the slicing operator :

In [8]:
```
# List slicing in Python

my_list = ['E','t','e','r','n','a','l','t','e','k']

# items from index 2 to index 4
print(my_list[2:7])  # n-1

# items from index 5 to end
print(my_list[5:])

# items beginning to end
print(my_list[:])
```

```
['e', 'r', 'n', 'a', 'l']
['a', 'l', 't', 'e', 'k']
['E', 't', 'e', 'r', 'n', 'a', 'l', 't', 'e', 'k']
```

In [9]:
```
my_list = ['E','t','e','r','n','a','l','t','e','k']

print(my_list[-3:-1]) # n+1
```

```
['t', 'e']
```

List Operations

List Contain Two types of Operations 1) Concatenation 2) Repetition

In [12]:
```
#concatenation

a = [1, 2, 3]
b = [4, 5, 6]
```

```
c = a + b
print(c)
```

```
[1, 2, 3, 4, 5, 6]
```

In [11]:
```
# Repetition

a = [3,6,9,12]

b = a * 7

print(b)
```

```
[3, 6, 9, 12, 3, 6, 9, 12, 3, 6, 9, 12, 3, 6, 9, 12, 3, 6, 9, 12, 3, 6,
9, 12]
```

Change List Items

To change the value of a specific item, refer to the index number:

In [13]:
```
lst = ["apple", "banana", "cherry"]
lst[1] = "Pineapple"
print(lst)
```

```
['apple', 'Pineapple', 'cherry']
```

In [19]:
```
lt = [2, 3.5, "Python", 4]
```

In [21]:
```
lt[2]= "Java"
lt[3] = 76

print(lt)
```

```
[2, 3.5, 'Java', 76]
```

In [24]:
```
#Change a Range of Item Values
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:4] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'kiwi', 'mango']
```

list methods

1) append():

Adds an element to the end of the list.

In [40]:
```
numbers = [1, 2, 3]
numbers.append(4)
print(numbers)
```

```
[1, 2, 3, 4]
```

2) extend():

Appends elements from another iterable to the end.

In [41]:
```
numbers = [1, 2, 3]
more_numbers = [4, 5, 6]
numbers.extend(more_numbers)
print(numbers)
```

```
[1, 2, 3, 4, 5, 6]
```

3) insert():

Inserts an element at a specified position.

In [4]:
```
numbers = [1, 2, 3]
numbers.insert(1, 99) # 1 indicates Index and 99 indicates value

numbers.insert(2, 9)
numbers.insert(-1, 9)
print(numbers)
```

```
[1, 99, 9, 2, 9, 3]
```

4) remove():

Removes the first occurrence of a value.

In [6]:
```
numbers = [1,3,6,2,3,2,4]
numbers.remove(3)
print(numbers)
```

```
[1, 6, 2, 3, 2, 4]
```

5) pop():

Removes and returns an element at a specified index.

In [10]:
```
numbers = [1, 2, 3, 4, 5, 7, 9, 4]
popped_element = numbers.pop(1)
print(numbers)
print(popped_element)
```

```
[1, 3, 4, 5, 7, 9, 4]
2
```

6) index():

Returns the index of the first occurrence of a value.

In [14]:
```
numbers = [10, 20, 30, 20, 40, 30, 30, 40]
index = numbers.index(40)
print(index)
```

```
4
```

count():

Returns the number of occurrences of a value.

In [16]:
```
numbers = [1, 2, 3, 2, 4, 2, 6, 7, 9, 6, 0, 2, 6, 4]
count = numbers.count(4)
print(count)
```

```
2
```

8) sort():

Sorts the list in ascending order.

In [26]:
```
numbers =[1, 2, 3, 2, 4, 2, 6, 7, 9, 6, 0, 2, 6, 4]
numbers.sort()
print(numbers)
```

```
[0, 1, 2, 2, 2, 2, 3, 4, 4, 6, 6, 6, 7, 9]
```

In [30]:
```
j = ['apple', 'cherry', 'pineaple','banana','support', 'dog']
j.sort()
print(j)
```

```
['apple', 'banana', 'cherry', 'dog', 'pineaple', 'support']
```

9) reverse():

Reverses the order of elements in the list.

In [28]:
```
numbers = [ 6, 0, 2, 6, 4]
numbers.reverse()
print(numbers)
```

```
[4, 6, 2, 0, 6]
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Syntax

newlist = [expression for item in iterable if condition == True]

Condition

The condition is like a filter that only accepts the items that valuate to True.

In [49]:
```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
['apple', 'banana', 'mango']
```

In [50]:
```
newlist = [x for x in fruits if x == "apple"]
```

In [51]:
```
print(newlist)
```

```
['apple']
```

Python Strings

In Python, a string is a sequence of characters.

We use single quotes or double quotes to represent a string in Python.

Strings are one of the most commonly used data types, and they are used to represent text.

Strings are immutable, meaning that once a string is created, you cannot modify its content directly.

However, you can perform various operations and use methods to manipulate and work with strings.

In [ ]:
```
# create a string using double quotes
str1 = "Python, sjljsfljsf , jlkfljljflk j, jljglgajk"

# create a string using single quotes
str2 = 'Python'
```

In [52]:
```
# If need any multi-line strings using triple qoutes
str3 = """This is a
multi-line string."""
print(str3)
```

```
This is a
multi-line string.
```

String Operations

1) Concatenation:

Combining two or more strings.

In [35]:
```
str1 = "Hello"
str2 = "World"
result = str1 +" "+ str2
print(result)
```

```
Hello World
```

2) Repetition:

Repeating a string multiple times.

In [38]:
```
str1 = "Python "
result = str1  * 3
print(result)
```

```
Python Python Python
```

3) Indexing:

Accessing individual characters in a string.

In [39]:
```
my_string = "Python"
print(my_string[0])
print(my_string[2])
```

```
P
t
```

4) Slicing:

Extracting a substring from a string.

In [40]:
```
my_string = "Hello, World!" #0123456789101112
substring = my_string[7:12]
print(substring)
```

```
World
```

Python - Format - Strings

1. Old Style Formatting (% operator):

Uses the % operator and placeholders.

In [21]:
```
name = "sai"
age = 30
formatted_string = "My name is %s and I am %d years old." % (name, age)
print(formatted_string)
```

```
My name is sai and I am 30 years old.
```

1. Format Method:

Uses the format() method with placeholders.

In [19]:
```
name = "sai"
age = 35
formatted_string = "My name is {} and I am {} years old.".format(name, age)
print(formatted_string)
```

```
My name is sai and I am 35 years old.
```

1. String Interpolation (f-strings - Python 3.6 and later):

Uses f-strings, where expressions inside curly braces {} are evaluated and replaced

In [24]:
```
name = "sai"
age = 25
```

```
formatted_string = f"My name is {name} and I am {age} years old."
print(formatted_string)
```

```
My name is sai and I am 25 years old.
```

String Methods

1) len():

Returns the length of the string.

In [26]:
```
my_string = "Python is easy language"
length = len(my_string)
print(length)
```

```
23
```

2) lower() / upper():

Converts the string to lowercase / uppercase.

In [27]:
```
my_string = "hello"

print(my_string.upper())
```

```
HELLO
```

In [28]:
```
my_string = "HELLO WORLD"
print(my_string.lower())
```

```
hello world
```

In [29]:
```
f = "PYTHON IS GOOD LANGUAGE"
print(f.lower())
```

```
python is good language
```

3) strip():

Removes leading and trailing whitespaces.

In [30]:
```
my_string = "   Python   "
stripped_string = my_string.strip()
print(stripped_string)
```

```
Python
```

In [37]:
```
f = "        Python   is                "
print(f.strip())
```

```
Python   is
```

4) replace():

Replaces a substring with another substring.

In [38]:
```
my_string = "Hello, World!"
new_string = my_string.replace("World", "Python") # (old value,new value)
```

```
print(new_string)
```

```
Hello, Python!
```

In [ ]:

5) count():

Returns the number of occurrences of a substring.

In [45]:
```
my_string = "python is a good language, python is easy language"
count = my_string.count("python")
print(count)
```

```
2
```

6) split():

Splits the string into a list of substrings.

In [46]:
```
my_string = "Python is fun and python is good to learn"
words = my_string.split()
print(words)
```

```
['Python', 'is', 'fun', 'and', 'python', 'is', 'good', 'to', 'learn']
```

7) join():

Joins a list of strings into a single string.

In [48]:
```
words = ['Python', 'is', 'fun', 'and', 'python','easy', 'to', 'learn']
joined_string = ' '.join(words)
print(joined_string)
```

```
Python is fun and python easy to learn
```

8) startswith() / endswith():

Checks if the string starts or ends with a specified substring.

In [52]:
```
my_string = "Hello, World!"
print(my_string.startswith("H"))
print(my_string.endswith("!"))
```

```
True
True
```

9) isalpha() / isnumeric() / isalnum():

Checks if the string consists of alphabetic characters / numeric characters / alphanumeric characters.

In [53]:
```
alpha_string = "Python"
numeric_string = "123"
alpha_numeric_string = "Python123"

print(alpha_string.isalpha())
```

```
print(numeric_string.isnumeric())
print(alpha_numeric_string.isalnum())
```

```
True
True
True
```

Python Tuples

A tuple is a data type that represents an ordered, immutable (unchangeable) sequence of elements.

Tuples are similar to lists, but the key difference is that tuples cannot be modified once they are created.

They are defined using parentheses () and elements are separated by commas.

Features of Python Tuple

Tuples are an immutable data type, meaning their elements cannot be changed after they are generated.

Each element in a tuple has a specific order that will never change because tuples are ordered sequences.

Creating a Tuple

In [54]:
```
#Empty Tuple:

empty_tuple = ()
```

In [55]:
```
#Tuple with Elements:
fruits = ("apple", "banana", "orange")
fruits
```

Out[55]:
```
('apple', 'banana', 'orange')
```

In [56]:
```
#Mixed Data Types:
mixed_tuple = (1, "apple", 3.14, True)
mixed_tuple
```

Out[56]:
```
(1, 'apple', 3.14, True)
```

In [61]:
```
#Single-element Tuple:

single_element_tuple = (42,)
single_element_tuple
```

Out[61]:
```
(42,)
```

Accessing of Tuples

Tuples support indexing and slicing, just like lists.

In [67]:
```
#           0        1          2
fruits = ("apple", "banana", "orange")
print(fruits[2])      #index
print(fruits[0:7])    #slicing
```

```
orange
('apple', 'banana', 'orange')
```

In [ ]:
```
```

Immutability:

Once a tuple is created, you cannot change its elements or add/remove elements.

In [77]:
```
fruits = ("apple", "banana", "orange")
# This will raise an error
fruits[2] = "pear"
#print(fruits)
fruits
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[77], line 3
      1 fruits = ("apple", "banana", "orange")
      2 # This will raise an error
----> 3 fruits[2] = "pear"
      4 #print(fruits)
      5 fruits

TypeError: 'tuple' object does not support item assignment
```

Tuple Operations

1) Concatenation:

Combining two tuples.

In [74]:
```
tuple1 = (1, 2, 3)
tuple2 = ('a', 'b', 'c')
result = tuple1 + tuple2
print(result)
```

```
(1, 2, 3, 'a', 'b', 'c')
```

2) Repetition:

Repeating a tuple.

In [75]:
```
tuple1 = (1, 2, 3)
result = tuple1 * 20
print(result)
```

```
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1,
2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2,
3, 1, 2, 3)
```

Tuple Methods

Tuples have limited methods compared to lists due to their immutability

1) count():

Returns the number of occurrences of a specified value.

In [ ]:
```
numbers = (1, 2, 2, 3, 4, 2, 4, 2, 9,4,8,2,9767,2)
count_of_2 = numbers.count(4)
print(count_of_2)
```

2) index():

Returns the index of the first occurrence of a specified value.

In [ ]:
```
numbers = (1, 2, 3, 4, 5)
index_of_3 = numbers.index(5)
print(index_of_3)
```

Python Dictionaries

A dictionary is a data structure that allows you to store and retrieve data in a key-value pair format.

Dictionaries are also known as associative arrays or hash maps in other programming languages.

They are defined using curly braces {} and consist of key-value pairs separated by colons.

Creating Dictionaries:

In [64]:
```
# Empty dictionary
empty_dict = {}

# Dictionary with key-value pairs
student = {
    'name': 'Reddy',
    'age': 25,
    'grade': 'A'
}
# dict1 = {Key:value,key:value,key:value}  ----> Syntax
```

```
# Dictionary with mixed data types
mixed_dict = {
    'name': 'Vamsi',
    'age': 30,
    'grades': [85, 90, 92]
}
mixed_dict
```

Out[64]:
```
{'name': 'Vamsi', 'age': 30, 'grades': [85, 90, 92]}
```

In [ ]:
```

```

In [65]:
```
print(empty_dict)
print(student)
print(mixed_dict)
```

```
{}
{'name': 'Eternaltek', 'age': 25, 'grade': 'A'}
{'name': 'Vamsi', 'age': 30, 'grades': [85, 90, 92]}
```

Accessing Values:

You can access the values in a dictionary using the keys.

In [1]:
```
# Dictionary with key-value pairs
student = {
    'name': 'Sai',
    'age': 25,
    'grade': 'A'
}
```

In [2]:
```
# Dictionary with mixed data types
mixed_dict = {
    'name': 'Vamsi',
    'age': 30,
    'grades': [85, 90, 92]
}
```

In [15]:
```
print(student['name'])  # Output: Sai
print(mixed_dict['grades'])  # Output: [85, 90, 92]
```

```
Sai
[85, 90, 92]
```

In [4]:
```
print(student['age'])
```

```
25
```

Modifying and Adding Entries:

Dictionaries are mutable, so you can modify existing entries or add new ones.

In [5]:
```
# Modifying an entry
student['age'] = 26
```

```
# Adding a new entry
student['gender'] = 'Female'
student['Qulification'] = 'B.Tech'
```

In [6]:
```
print(student)
```

```
{'name': 'Sai', 'age': 26, 'grade': 'A', 'gender': 'Female', 'Qulification': 'B.Tec
h'}
```

Dictionary Methods:

1) get():

Returns the value for a specified key. It allows you to provide a default value if the key is not found.

In [11]:
```
g = student.get('grade', 'Not specified')
gender = student.get('gender', 'Not specified')
h = student.get('location', 'Not Prividied')
#g = student.get('exper')
```

In [12]:
```
print(g)
print(gender)
print(h)
```

```
A
Female
Not Prividied
```

2) keys():

Returns a view of all the keys in the dictionary.

In [13]:
```
keys = student.keys()
print(keys)
```

```
dict_keys(['name', 'age', 'grade', 'gender', 'Qulification'])
```

3) values():

Returns a view of all the values in the dictionary.

In [14]:
```
values = student.values()
print(values)
```

```
dict_values(['Sai', 26, 'A', 'Female', 'B.Tech'])
```

4) items():

Returns a view of all key-value pairs as tuples.

In [15]:
```
items = student.items()
print(items)
```

```
dict_items([('name', 'Sai'), ('age', 26), ('grade', 'A'), ('gender', 'Female'), ('Qul
ification', 'B.Tech')])
```

5) pop():

Removes and returns the value for a specified key.

In [16]:
```
# Dictionary with mixed data types
mixed_dict = {
    'name': 'Vamsi',
    'age': 30,
    'grades': [85, 90, 92]
}
```

In [17]:
```
age =mixed_dict.pop('age')
print(age)
#print(student)
```

```
30
```

In [18]:
```
name = mixed_dict.pop()
name
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[18], line 1
----> 1 name = mixed_dict.pop()
      2 name

TypeError: pop expected at least 1 argument, got 0
```

6) popitem():

Removes and returns the last inserted key-value pair as a tuple.

In [19]:
```
last_item = student.popitem()
print(last_item)
```

```
('Qulification', 'B.Tech')
```

7) update():

Updates the dictionary with elements from another dictionary or from an iterable of key-value pairs.

In [20]:
```
# Dictionary with key-value pairs
student = {
    'name': 'Reddy',
    'age': 25,
    'grade': 'A'
}
student
```

Out[20]:
```
{'name': 'Reddy', 'age': 25, 'grade': 'A'}
```

In [21]:
```
new_data = {'age': 27, 'grade': 'A+'}
student.update(new_data)
```

```
print(student)
```

```
{'name': 'Reddy', 'age': 27, 'grade': 'A+'}
```

Python Sets

In Python, a set is a collection of unique elements, and it is defined using curly braces {} and sep by , (comma)

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

Sets are unordered, mutable, and do not allow duplicate elements.

Sets provide a variety of methods for performing common set operations like union, intersection, difference, and more

Creating Sets:

In [22]:
```
# Creating an empty set
#empty_set = set()

# Creating a set with elements
fruits = {'apple', 'banana', 'orange'}

# Converting a list to a set
numbers = str([1, 2, 3, 4, 5])
```

In [23]:
```
numbers
```

Out[23]:
```
'[1, 2, 3, 4, 5]'
```

In [84]:
```
# Creating an empty set
empty_set = set()
empty_set
```

Out[84]:
```
set()
```

In [24]:
```
#Duplicate Items in a Set
numbers = {2, 4, 6, 6, 2, 8, 8, 3, 2, 9, 5}
print(numbers)
```

```
{2, 3, 4, 5, 6, 8, 9}
```

Add and Update Set Items in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing.

Set data type does not support it.

1) Add Items to a Set in Python

In Python, we use the add() method to add an item to a set.

In [26]:
```
numbers = {21, 34, 54, 12}

print('Initial Set:',numbers)

# using add() method
numbers.add(78)

print('Updated Set:', numbers)
```

```
Initial Set: {34, 12, 21, 54}
Updated Set: {34, 12, 78, 21, 54}
```

In [ ]:
```

```

2) Update Python Set

The update() method is used to update the set with items other collection types (lists, tuples, sets, etc).

In [29]:
```
companies = {'microsoft', 'Technologies'}
tech_companies = ['apple', 'google', 'apple']

companies.update(tech_companies)

print(companies)
```

```
{'apple', 'Technologies', 'google', 'microsoft'}
```

In [ ]:
```

```

Remove an Element from a Set

We use the discard() method to remove the specified element from a set.

In [89]:
```
languages = {'Swift', 'Java', 'Python'}

print('Initial Set:',languages)

# remove 'Java' from a set
removedValue = languages.discard('Java')
# remove 'Java' from a set
#removedValue = languages.remove('Java')
print('Set after remove():', languages)
```

```
Initial Set: {'Python', 'Swift', 'Java'}
Set after remove(): {'Python', 'Swift'}
```

In [30]:
```
#finding the len of set

even_numbers = {2,4,6,8,10,12,14,12}
print('Set:',even_numbers)

# find number of elements
print('Total Elements:', len(even_numbers))
```

```
Set: {2, 4, 6, 8, 10, 12, 14}
Total Elements: 7
```

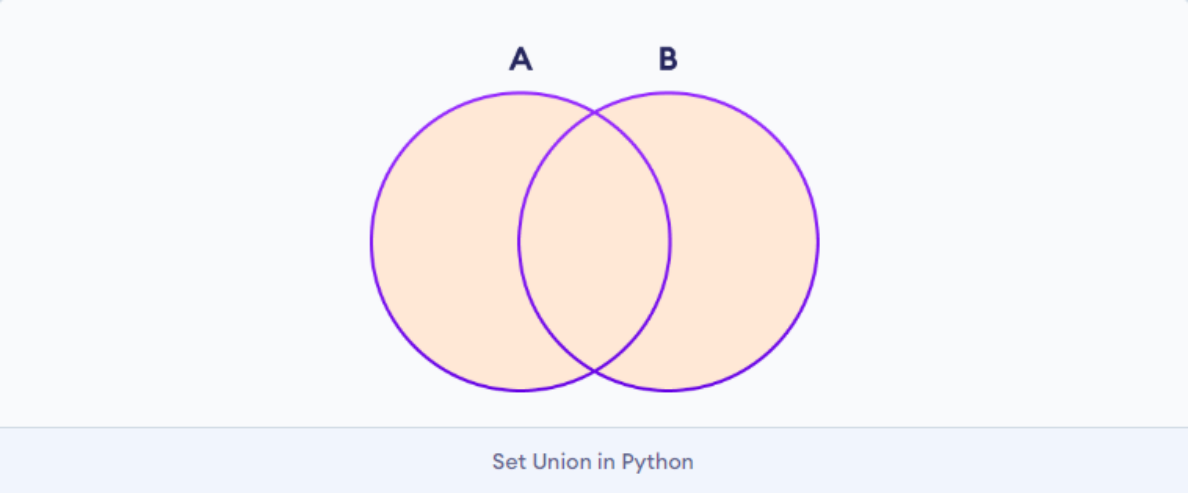
Python Set Operations

Python Set provides different built-in methods to perform mathematical set operations like union, intersection, subtraction, and symmetric difference.

1) Union of Two Sets

The union of two sets A and B include all the elements of set A and B.

Combines elements from two sets, removing duplicates.



Set Union in Python

In [91]:
```python
# first set
c = {1, 3, 5}

# second set
d = {0, 2, 4}

# perform union operation using |
print('Union using |:', c | d)

# perform union operation using union()
print('Union using union():', c.union(d))
```

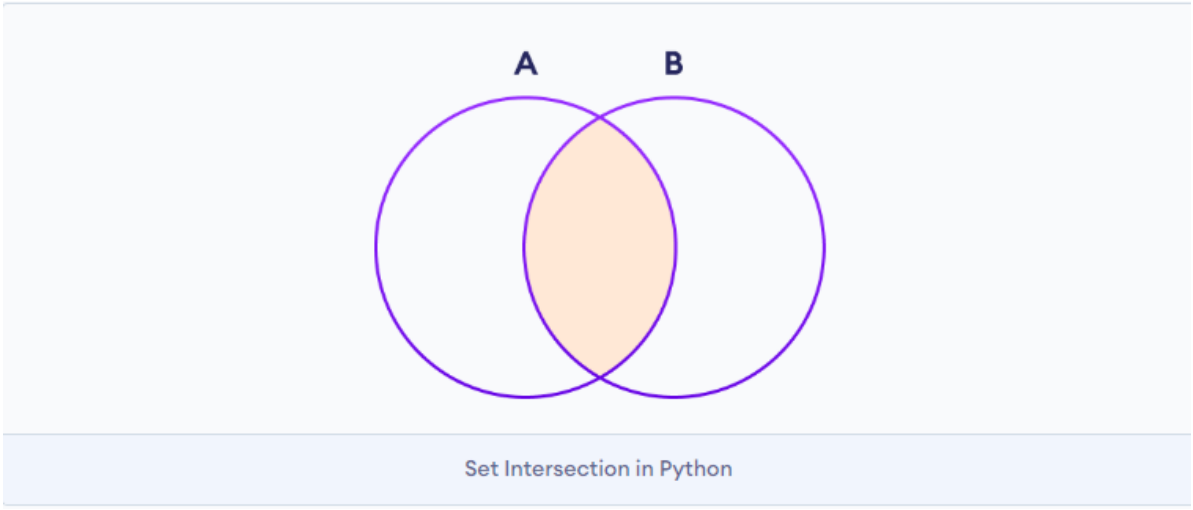
```
Union using |: {0, 1, 2, 3, 4, 5}
Union using union(): {0, 1, 2, 3, 4, 5}
```

In [ ]:
```python
A.union(B)
```

2) Set Intersection

The intersection of two sets A and B include the common elements between set A and B.

Returns common elements between two sets.



Set Intersection in Python

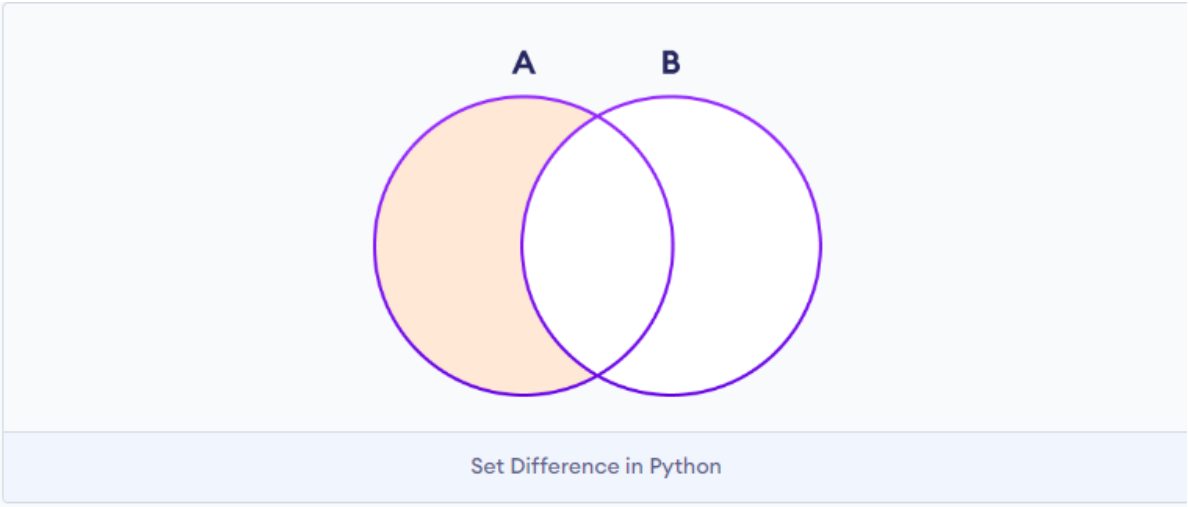
In [4]:
```
# first set
A = {1, 3, 5, 8, 4, 0}

# second set
B = {1, 2, 3, 0, 5, 4}

# perform intersection operation using &
print('Intersection using &:', A & B)

# perform intersection operation using intersection()
print('Intersection using intersection():', A.intersection(B))
```

```
Intersection using &: {0, 1, 3, 4, 5}
Intersection using intersection(): {0, 1, 3, 4, 5}
```

3) Difference between Two Sets

The difference between two sets A and B include elements of set A that are not present on set B.

Returns elements that are in the first set but not in the second.



Set Difference in Python

In [6]:
```
# first set
aA = {2, 3, 5, 9, 0, 4}

# second set
```

```
B = {1, 2, 6, 4, 7}

# perform difference operation using -
print('Difference using -:', A - B)

# perform difference operation using difference()
print('Difference using difference():', A.difference(B))
```

```
Difference using -: {0, 9, 3, 5}
Difference using difference(): {0, 9, 3, 5}
```

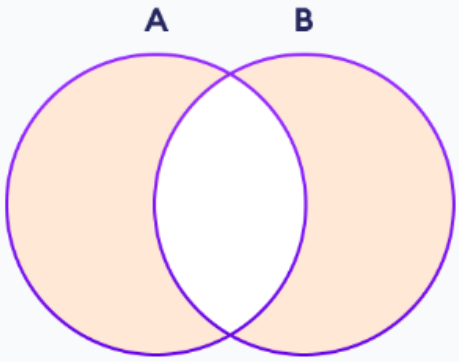
In [7]:
```
# perform difference operation using -
print('Difference using -:', B-A)

# perform difference operation using difference()
print('Difference using difference():', B.difference(A))
```

```
Difference using -: {1, 6, 7}
Difference using difference(): {1, 6, 7}
```

4) Set Symmetric Difference

The symmetric difference between two sets A and B includes all elements of A and B without the common elements.

Returns elements that are in either of the sets, but not in both.



Set Symmetric Difference in Python

In [9]:
```
# first set
A = {2, 3, 5, 7, 8, 4}

# second set
B = {1, 2, 6, 4, 9}

# perform difference operation using ^
print('using ^:', A ^ B)

# using symmetric_difference()
print('using symmetric_difference():', A.symmetric_difference(B))
```

```
using ^: {1, 3, 5, 6, 7, 8, 9}
using symmetric_difference(): {1, 3, 5, 6, 7, 8, 9}
```

In [ ]:

Python Conditions Statements

In Python, conditional statements are used to control the flow of a program based on certain conditions.

The if, else, and elif (short for "else if") keywords are used for creating conditional structures.

1) Python if Statement

An if statement executes a block of code only if the specified condition is met.

Syntax

if condition:

```
    # body of if statement
```

Here, if the condition of the if statement is:

True - the body of the if statement executes.

False - the body of the if statement is skipped from execution.

In [ ]:

In [14]:
```
number = 43
# check if number is greater than 0
if number > 0:  #condition --- if condition is TRue it will the return the body statem
    print('Number is positive')
    print("Python")

#print('This statement always executes')
```

```
Number is positive
Python
```

In [ ]:

In [ ]:

In [ ]:

Python else Statement

An if statement can have an optional else clause.

The else statement executes if the condition in the if statement evaluates to False.

Syntax

if condition:

```
    # body of if statement
```

else:

```
    # body of else statement
```

Here, if the condition inside the if statement evaluates to

True - the body of if executes, and the body of else is skipped.

False - the body of else executes, and the body of if is skipped

In [19]:
```
number = 56

if number > 0:
    print('Positive number')

else:
    print('Negative number')

#print('This statement always executes')
```

```
Positive number
```

In [ ]:
```

```

python elif condition

syntax: if condition:

```
    #body statement
```

elif condition:

```
    #body statements
```

else:

```
    #body statements
```

In [27]:
```
number = -45.7

if number > 0:
    print("Positive number")

elif number <0:
    print('Negative number')

else:
    print('Zero')

#print('This statement is always executed')
```

```
Negative number
```

1) static type input: intital assgin by developer or code writer 2) dymanic type input: inputs enter by user

In [33]:
```
number = int(input("Enter your lucky number: ")) #number you to mention int()

if number > 0:
    print("your lucky number is Positive")
elif number < 0:
    print("your lucky is negative")
else:
    print("your lucky number is Zero")
```

```
Enter your lucky number: 6
your lucky number is Positive
```

In [35]:
```
name = input("Enter you name: ")
print(name)
```

```
Enter you nameVamsi Reddy
Vamsi Reddy
```

Python Nested if Statements

It is possible to include an if statement inside another if statement.

In [ ]:
```
number= int(input("Enter your number: "))

# outer if statement
if number >= 0: #-8 >= 0
    # inner if statement
    if number == 0:
        print('Number is 0')
    # inner else statement
    else:
        print('Number is positive')

# outer else statement
else:
    print('Number is negative')
```

In [ ]:
```
#example on conditional statements

#Example 2: Grading System
```

```
score = int(input("Enter your score:  "))

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
elif score >= 50:
    grade = 'E'
else:
    grade = 'F'

print(f"Your grade is {grade}")
```

In [4]:
```
# Three nested if conditions example

# Sample values for demonstration
a = 15
b = 25
c = 25

if a > 10:
    print("a is greater than 10")

    if b > 20:
        print("b is greater than 20")

        if c > 30:
            print("c is greater than 30")
        else:
            print("c is not greater than 30")

    else:
        print("b is not greater than 20")

else:
    print("a is not greater than 10")
```

```
a is greater than 10
b is greater than 20
c is not greater than 30
```

In [ ]:
```

```

Python while Loop

Python while loop is used to run a block code until a certain condition is met.

The syntax of while loop is:

while condition:

```
    # body of while loop
```

Here,

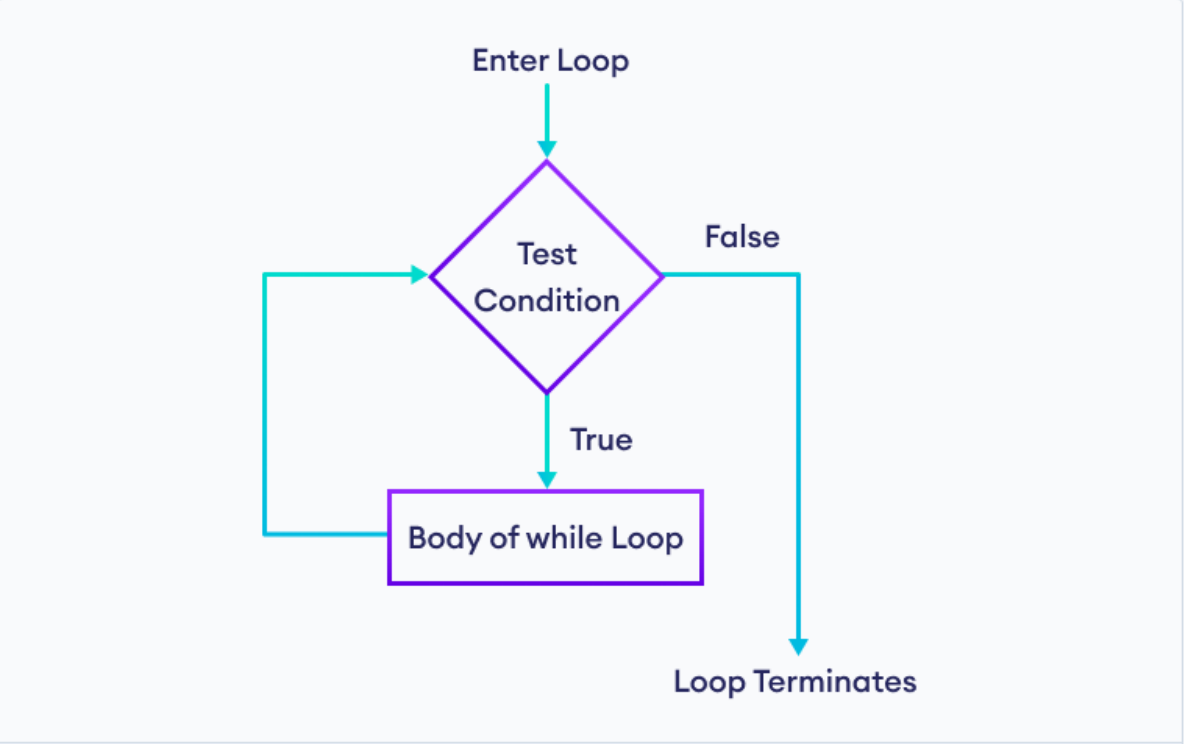
-> A while loop evaluates the condition

-> If the condition evaluates to True, the code inside the while loop is executed.

-> condition is evaluated again.

-> This process continues until the condition is False.

-> When condition evaluates to False, the loop stops.



In [1]:
```
# program to display numbers from 1 to 5

# initialize the variable
i = 20


# while loop from i = 1 to 5
while i <= 100:
    print(i)
    i = i * 2 #increment
```

```
20
40
80
```

In [ ]:
```

```

Python for Loop:

The for loop is used for iterating over a sequence (that is either a list, tuple, dictionary, string, or range).

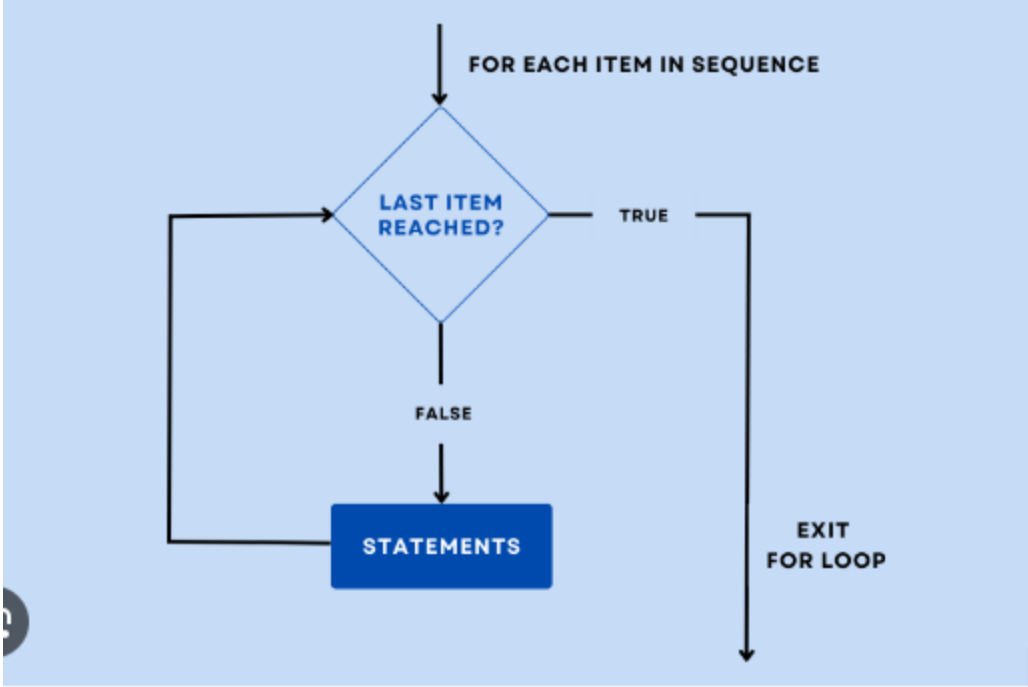
The syntax of a for loop is:

for val in sequence:

```
    # statement(s)
```

Here, val accesses each item of the sequence on each iteration.

The loop continues until we reach the last item in the sequence.



In [103…

```
language = 'Python'

# iterate over each character in language
for x in language:
    print(x)
```

```
P
y
t
h
o
n
```

In [12]:

```
#Example 1: Iterate Over a List

fruits = ['apple', 'banana', 'orange']

for fruit in fruits:
    print(fruit)
```

```
apple
banana
orange
```

In [1]:
```
#Example 2: Iterate Over a Range
for number in range(19, 50):
    print(number)
```

```
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
```

In [2]:
```
#Example 3: Iterate Over a String
word = "Eternaltek"

for char in word:
    print(char)
```

```
E
t
e
r
n
a
l
t
e
k
```

In [4]:
```
#Example 4: Iterate Over a Dictionary
student_grades = {'Vamsi': 90, 'Reddy': 85, 'Kumar': 92, 'shiva': 98}
```

```
for name, grade in student_grades.items():
    print(f"{name}'s grade is {grade}")
```

```
Vamsi's grade is 90
Reddy's grade is 85
Kumar's grade is 92
shiva's grade is 98
```

In [ ]:

Control Statements

In Python, pass, break, and continue are control flow statements used within loops.

They provide ways to control the execution of a loop or to handle certain situations.

Here's an explanation of each, along with examples:

In [ ]:

1. pass:

The pass statement is a no-operation statement.

It is a placeholder where syntactically some code is required, but no action needs to be taken.

In [104…

```
for i in range(5): #output = 0,1,,2,3,4
    if i == 2:
        pass  # do nothing for i = 2
    else:
        print(i)
```

```
0
1
3
4
```

In this example, when i is equal to 3, the pass statement is executed, and the loop continues without doing anything.

It is often used when a statement is syntactically required, but you want to skip its execution.

In [ ]:

2. break:

The break statement is used to exit a loop prematurely.

When a break statement is encountered, the loop terminates immediately, and the control is transferred to the statement immediately following the loop.

In [6]:
```
for i in range(10): #0123456789
    if i == 15:
        break  # exit the loop when i reaches 5
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

In this example, the loop will print values from 0 to 4. When i becomes 5, the break statement is encountered, and the loop is terminated.

In [ ]:
```

```

3. continue:

The continue statement is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration.

In [9]:
```
for i in range(25):
    if i == 5:
        continue  # skip the rest of the code for i = 7
    elif i == 7:
        continue
    elif i == 17:
        continue
    else:
        print(i)
```

```
0
1
2
3
4
6
8
9
10
11
12
13
14
15
16
18
19
20
21
22
23
24
```

In [ ]:

In this example, the loop will print values from 0 to 4, skipping the printing step when i is equal to 2 due to the continue statement.

In [ ]:
```
def gin():
```

Python Functions

A function is a block of code that performs a specific task.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Types of function

There are two types of function in Python programming:

Standard library functions - These are built-in functions in Python that are available to use.

User-defined functions - We can create our own functions based on our requirements.

The syntax to declare a function is:

def function_name(arguments):

```
    # function body

    return
```

Here,

def - keyword used to declare a function

function_name - any name given to the function

arguments - any value passed to function

return (optional) - returns value from a function

In [1]:
```
#simple function
def greet():
    print('Hello World!')
    print('sdjhfka')

print()
```

Calling a Function in Python

In the above example, we have declared a function named greet().

Now, to use this function, we need to call it.

Here's how we can call the greet() function in Python.

In [11]:
```
def greet(name):
    """This function greets the person passed in as a parameter."""
    print(f"Hello, {name}!")

# Calling the function
greet("Reddy")  #function_name(parameter)
greet("F")
```

```
Hello, Reddy!
Hello, F!
```

In [ ]:
```

```

In [9]:
```
def vote(name,age):
    print(f'your name is  {name} and your {age} is Eogluta vote')

vote("reddy", 23)
```

```
your name is  reddy and your 23 is Eogluta vote
```

Python Function Arguments

a function can also have arguments. An argument is a value that is accepted by a function. For example,

In [12]:
```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum: ",sum)

# function call with two values
add_numbers(4453, 4)
```

```
Sum:  4457
```

The return Statement in Python

A Python function may or may not return a value. If we want our function to return some value to a function call, we use the return statement.

def add_numbers(): ... return sum Here, we are returning the variable sum to the function call.

In [13]:
```
# function definition
def find_square(num):
    result = num * num
    return result

# function call
square = find_square(3)

print('Square:',square)
```

```
Square: 9
```

Python Library Functions

In Python, standard library functions are the built-in functions that can be used directly in our program. For example,

print() - prints the string inside the quotation marks

sqrt() - returns the square root of a number

pow() - returns the power of a number

These library functions are defined inside the module. And, to use them we must include the module inside our program.

For example, sqrt() is defined inside the math module.

In [14]:
```
import math

# sqrt computes the square root
square_root = math.sqrt(4)

print("Square Root of 4 is",square_root)

# pow() comptes the power
power = pow(2, 3)

print("2 to the power 3 is",power)
```

```
Square Root of 4 is 2.0
2 to the power 3 is 8
```

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

lambda arguments : expression

In [16]:
```
x = lambda a : a + 10
print(x(55)) # print(x(a))
```

```
65
```

In [17]:
```
def myfunc(n):
    return lambda a : a * n

mydouble = myfunc(5)

print(mydouble(11))
```

```
55
```

Global and Local Variables in Python

Python Global variables are those which are not defined inside any function and have a global scope whereas Python local variables are those which are defined inside a function and their scope is limited to that function only.

In other words, we can say that local variables are accessible only inside the function in which it was initialized whereas the global variables are accessible throughout the program and inside every function.

In [18]:
```
#Creating local variables in Python
def f():

    # local variable
    s = "I love Eternaltek"
    print(s)


# Driver code
f()
```

```
I love Eternaltek
```

In [19]:
```
def rt():
    pass

rt()
```

In [ ]:
```
#Can a local variable be used outside a function?

#If we will try to use this local variable outside the function then let's see what wi
```

In [23]:
```
def f():

        # local variable
        s = "I love Eternaltek"
        print("Inside Function:", s)

# Driver code
f()
print(s)
```

```
Inside Function: I love Eternaltek
rtyr
```

Python Global Variables

These are those which are defined outside any function and which are accessible throughout the program, i.e.,

inside and outside of every function. Let's see how to create a Python global variable.

Create a global variable in Python

Defining and accessing Python global variables

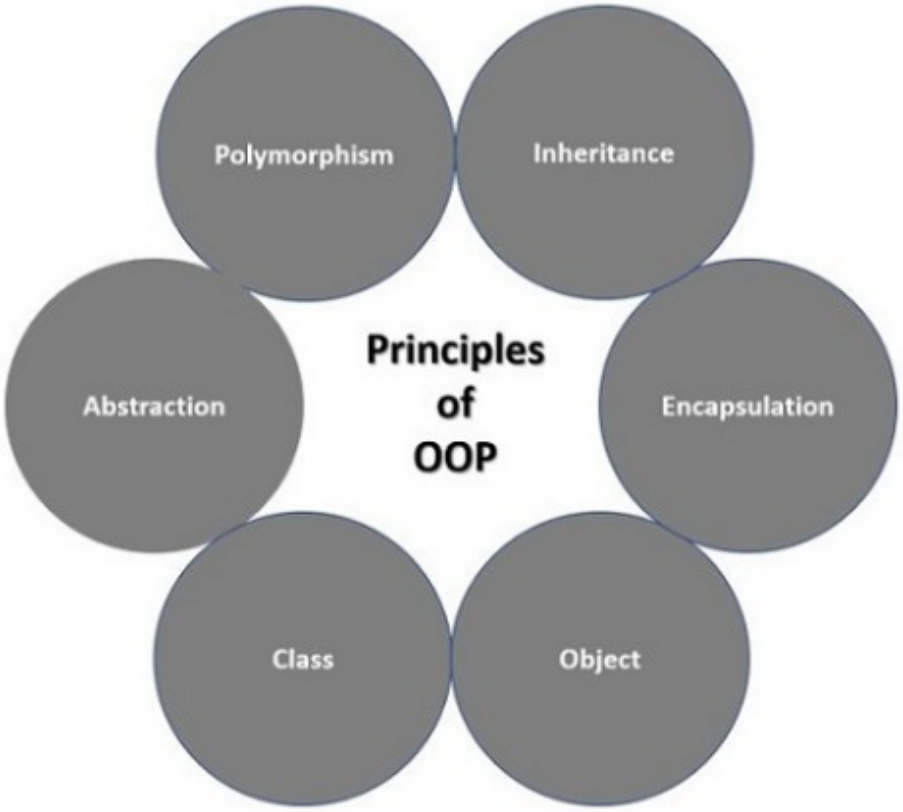
In [24]:
```
# This function uses global variable s
def f():
    print("Inside Function", s)

# Global scope
s = "I love Eternaltek"
f()
print("Outside Function", s)
```

```
Inside Function I love Eternaltek
Outside Function I love Eternaltek
```

In [ ]:
```

```

Python Object Oriented Programming

Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOP) through the use of objects and classes.



Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

In [3]:
```
class MyClass:
    x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

- Object is physical entity
- we can create any No. of object for class
- Memory is allocated when we create object for class

In [2]:
```
obj1 = MyClass()
print(obj1.x)
```

```
5
```

In [8]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Woof!")

# Creating objects of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Accessing object attributes and calling methods
print(f"{dog1.name} is {dog1.age} years old.")
print(f"{dog2.name} is {dog2.age} years old.")
dog1.bark()
```

```
Buddy is 3 years old.
Max is 5 years old.
Woof!
```

The self Parameter

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 does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

In [11]:
```
class self():
    def __init__(self):
        print("Hello world")


obj = self()
```

```
Hello world
```

The **init**() Function

The **init** method (pronounced "dunder init") is called the constructor.

It is automatically executed when an object is created from a class.

Its purpose is to initialize the attributes of the object.

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

```
# Creating a Dog object
my_dog = Dog(name="Buddy", age=3)
```

In [ ]:

In this example, when you create a Dog object (my_dog), the **init** method is automatically called, and it initializes the name and age attributes of the object.

str Method:

The **str** method is used to provide a human-readable string representation of the object.

It is called when the str() function is used on an object or when print() is called with an object.

In [13]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, {self.age} years old"

# Creating a Dog object
my_dog = Dog(name="Buddy", age=3)

# Using str() or print() on the object
#print(str(my_dog))  # Output: Buddy, 3 years old
print(my_dog)       # Output: Buddy, 3 years old
```

```
Buddy, 3 years old
```

In this example, the **str** method is defined to return a formatted string representing the dog's name and age.

When str() or print() is called on the my_dog object, the **str** method is invoked, providing a readable output.

In [ ]:

Python Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it.

The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

In [14]:
```
# base class
class Animal:

    def eat(self):
```

```
        print( "I can eat!")

    def sleep(self):
        print("I can sleep!")

# derived class
class Dog(Animal):

    def bark(self):
        print("I can bark! Woof woof!!")

class Cat(Animal):

    def Mayoo(self):
        print(" I can Mayoo! Mayoo!")

# Create object of the Dog class
dog1 = Dog()

# Calling members of the base class
dog1.eat()
dog1.sleep()

# Calling member of the derived class
dog1.bark();

obj2 = Cat()

obj2.eat()
obj2.sleep()
obj2.Mayoo()
```

```
I can eat!
I can sleep!
I can bark! Woof woof!!
I can eat!
I can sleep!
 I can Mayoo! Mayoo!
```

In [ ]:

Python Encapsulation

Encapsulation is one of the key features of object-oriented programming.

Encapsulation refers to the bundling of attributes and methods inside a single class.

wrapping data and methods that work with data in one unit.

This also helps to achieve data hiding.

In Python, we denote private attributes using underscore as the prefix i.e single _(Protected) or double __ (Private) and Public. For example,

In [ ]:

In [5]:
```
class Demo():
    def __init__(self,a,b):
        self.__a = a # Private
        self._b = b # PRotected

class Demo1(Demo):
    def output(self):
        print(self._b)

d = Demo1(3,4)
d.output()
```

```
4
```

In [ ]:
```

```

Polymorphism

Polymorphism is another important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

In [ ]:
```
#class Polygon:
    # method to render a shape
    #def render(self):
    #    print("Rendering Polygon...")

#class Square(Polygon):
    # renders Square
   # def render(self):
    #    print("Rendering Square...")

#class Circle(Polygon):
    # renders circle
   # def render(self):
     #   print("Rendering Circle...")

# create an object of Square
#s1 = Square()
#s1.render()

# create an object of Circle
#c1 = Circle()
#c1.render()
```

In [9]:
```
class sum1():
    def add(self,a,b):
        print(a+b)

obj = sum1()

print(obj.add(3,5))
print(obj.add('a','b'))
```

```
8
None
ab
None
```

Abstraction

Hiding the unnecessary details.

Abstraction allows you to focus on what an object does rather than how it achieves its functionality.

In [ ]:
```python
class Car:
    def start_engine(self):
        pass  # Abstract method, implementation details hidden

    def drive(self):
        pass  # Abstract method, implementation details hidden

    def stop_engine(self):
        pass  # Abstract method, implementation details hidden
```

In [ ]:
```python

```

Key Points to Remember:

Object-Oriented Programming makes the program easy to understand as well as efficient.

Since the class is sharable, the code can be reused.

Data is safe and secure with data abstraction.

Polymorphism allows the same interface for different objects, so programmers can write efficient code.

In [ ]:
```python

```

Python File I/OPython File Operation

A file is a container in computer storage devices used for storing data.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1) Open a file 2) Read or write (perform operation) 3) Close the file

Opening Files in Python

In Python, we use the open() method to open files.

To demonstrate how we open files in Python, let's suppose we have a file named test.txt with the following content.

In [108…
```
# open file in current directory
file1 = open("test.txt")
```

In [109…
```
#By default, the files are open in read mode (cannot be modified). The code above is e

file1 = open("test.txt", "r")
```

Different Modes to Open a File in Python

Mode	Description
r	Open a file for reading. (default)
w	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Open a file for exclusive creation. If the file already exists, the operation fails.
a	Open a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Open in text mode. (default)
b	Open in binary mode.
+	Open a file for updating (reading and writing)

Reading Files in Python

After we open a file, we use the read() method to read its contents. For example,

In [111…
```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)
```

```
Hello, World!

i had completed python class
```

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file.

It is done using the close() method in Python. For example,

In [112…
```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)

# close the file
file1.close()
```

```
Hello, World!

i had completed python class
```

Exception Handling in Files

If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

The try...except block is used to handle exceptions in Python. Here's the syntax of try...except block: try: # code that may cause exception except: # code to run when exception occurs

In [113…
```
try:
    file1 = open("test.txt", "r")
    read_content = file1.read()
    print(read_content)

finally:
    # close the file
    file1.close()
```

```
Hello, World!

i had completed python class
```

In [ ]:
```

```

In [115…
```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

```
Error: Denominator cannot be 0.
```

In [ ]: