1. Introduction to Python -

- Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by **Guido van Rossum** during **1985 - 1990**. Python source code is also available under the GNU General Public License (GPL) and we are allowed to come up with our own versions as its open source.
- Python is named after a TV Show called **Monty Python's Flying Circus** and not after Python, the snake.

1.1 Versions:

Version	Released Year
Python 1.0	1994
Python 2.0	2000
Python 3.0	2008

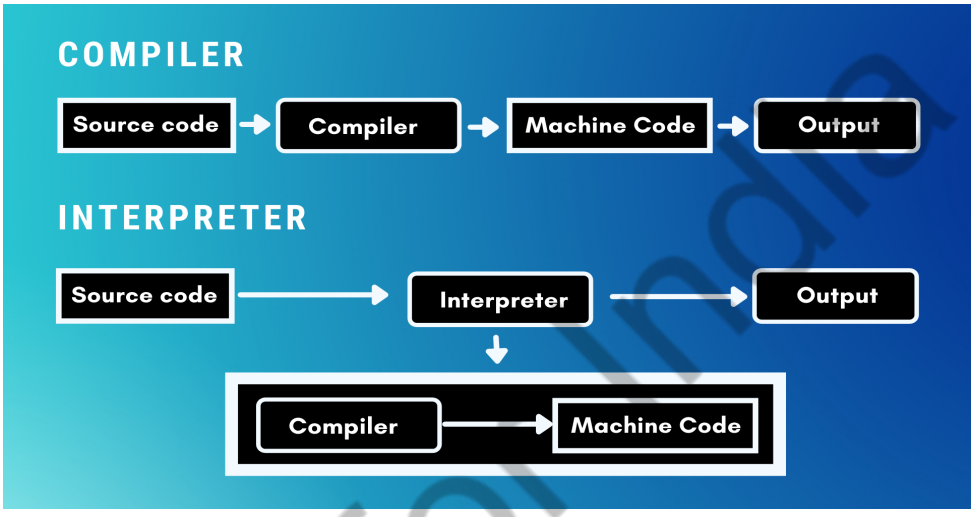
Currently Python version 3.10 is active.

1.2 Why python?

1. Simple Easy-to-use Syntax like simple English language terms
2. Compatible in all Operating systems like Windows, Mac, Linux, Raspberry Pi, etc
3. Code readability
4. Length of code is very less compared to other programming languages
5. Complex problems or algorithms can be simplified
6. Python is an interpreter language, which means that code may be run as soon as it is written. As a result, prototyping may be done quickly
7. Python can be approached in three ways: procedural, object-oriented, and functional
8. Wide Range of Libraries(200+) and Frameworks are available for advanced tasks

Few others -

- **Python** is a general-purpose and high-level programming language.
- You can **use Python** for developing desktop GUI applications, websites and web applications.
- Also, **Python**, as a high-level programming language, allows you to focus on the core functionality of the application by taking care of common programming tasks.
- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.



- **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports an Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** − Python is a great language for beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

1.3 Comparison among programming languages:

	C	C++	JAVA	PYTHON
Language Type	Procedure Oriented Language	Object Oriented Language	Object Oriented Language	Both Procedural & Object Oriented Language
Building Block	Function Driven	Object Driven	Both Object and Class Driven	Function, Object and Class Driven
Extension	.c	.cpp	.java	.py
Platform	Dependent	Independent	Independent	Independent
Comment Style	/* */	// Single Line /* */ Multi-Line	// Single Line /* */ Multi-Line	# Single Line """ """ Multi Line
Translation Type	Compiled	Compiled	Compiled & Interpreted	Interpreted
Database Connectivity	not supported	not supported	Supported	Supported
Representing Block of Statements	{ }	{ }	{ }	Indentation
Declaring Variables	Required	Required	Required	Not Required
Applications	Compilers , Interpreters , Embedded Programming etc.	Simple Desktop Applications, Embedded Systems etc.	Desktop GUI, Mobile, Web, Gaming etc.	Desktop, Web, Gaming, Network Programming etc.
IDE	Turbo C, Code Blocks	Turbo C++, Code Blocks	Eclipse , NetBeans etc	PyCharm , PyDev, IDLE, Jupyter No... etc

Full screen (f)

1.4 Applications:

- Web & Inernet Development
- Desktop GUI Applications
- Software Development
- Game Development
- AI & ML
- Data Analytics
- Business Applications
- Image Processing Applications
- Database access and handling

Environment setup / Installation

Python shell can be installed from the official website of python -

https://www.python.org/downloads/

- On the downloads page, Select the **Operating system** that you want to download
- You can select the required version of python
- Select your storage (32 Bit/ 64 Bit)

- Download the installation setup file

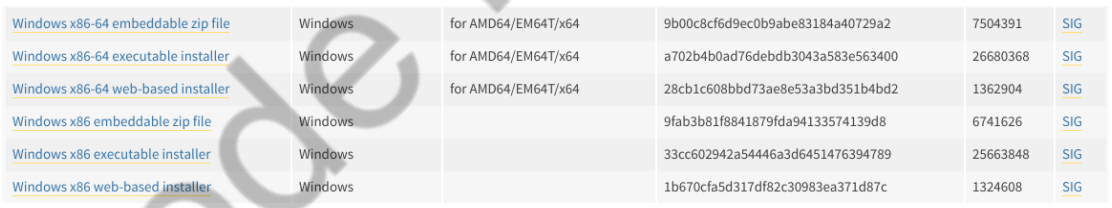
NOTE: Ubuntu Operating system has inbuilt python. No need to download the setup like other OS.

Follow the steps below :

Download Python Latest Version from python.org

- The first and foremost step is to open a browser and open **https://www.python.org/downloads/windows/**

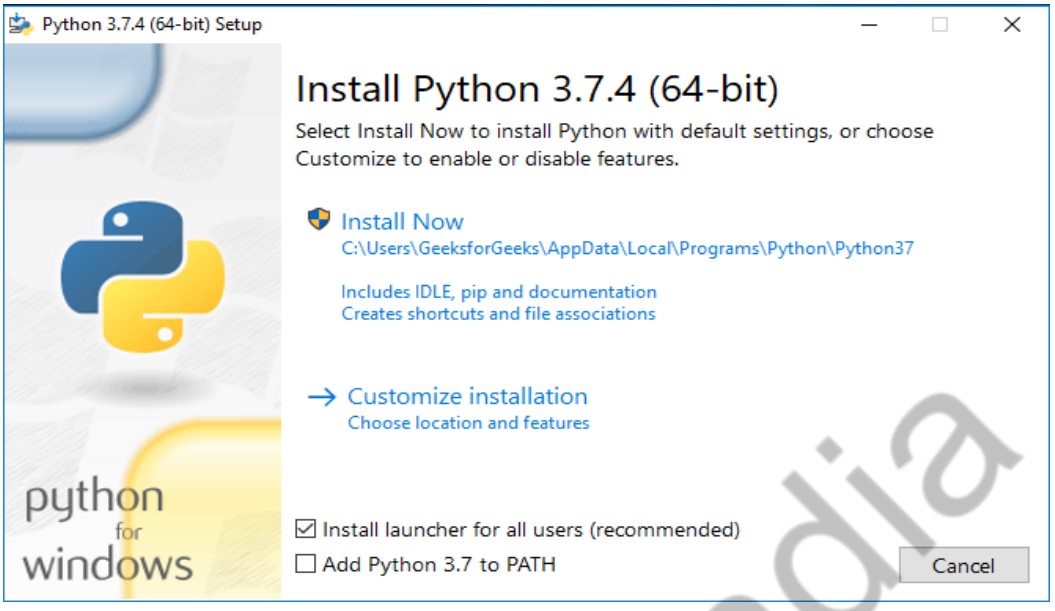
- Underneath the Python Releases for Windows find the Latest Python 3 Release
- On this page move to Files and click on Windows x86-64 executable installer for 64-bit or Windows x86 executable installer for 32-bit.



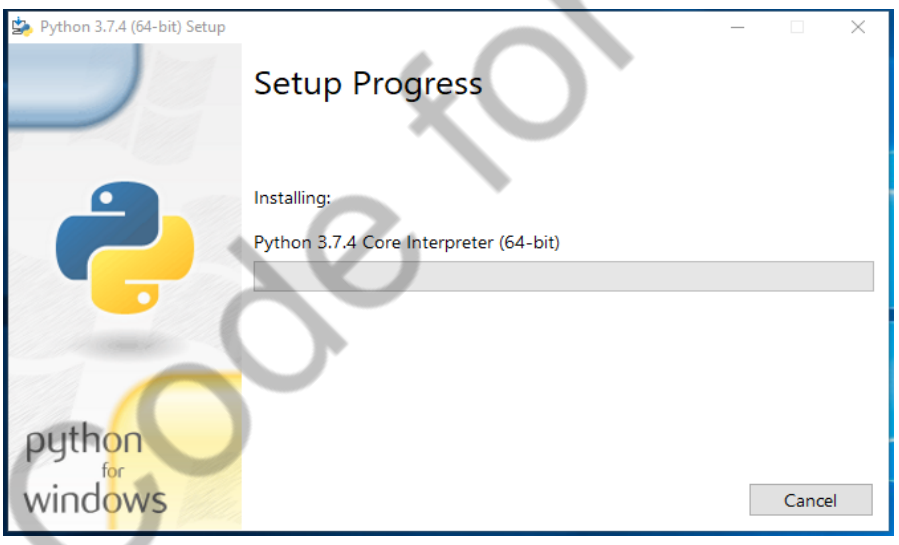
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	9b00c8cf6d9ec0b9abe83184a40729a2	7504391	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	a702b4b0ad76debdb3043a583e563400	26680368	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	28cb1c608bbd73ae8e53a3bd351b4bd2	1362904	SIG
Windows x86 embeddable zip file	Windows		9fab3b81f8841879fda94133574139d8	6741626	SIG
Windows x86 executable installer	Windows		33cc602942a54446a3d6451476394789	25663848	SIG
Windows x86 web-based installer	Windows		1b670cfa5d317df82c30983ea371d87c	1324608	SIG

- Install Python 3.7.4 Latest Version on Windows

- Run the Python Installer from the downloads folder

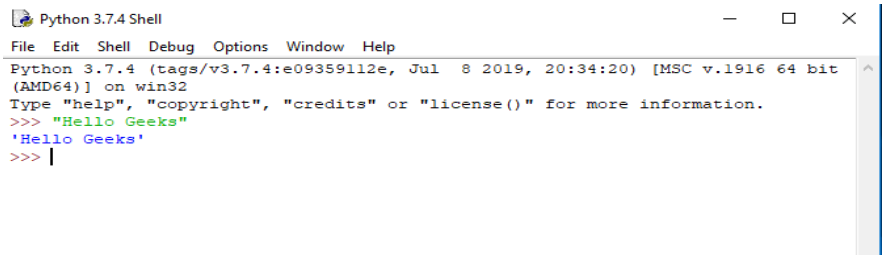


- Make sure to mark Add Python 3.7 to PATH otherwise you will have to do it explicitly.
- It will start installing python on windows.



- After installation is complete click on Close.

  Python is installed. Now go to windows and type IDLE or python.
- **Kudos! You are ready to Master Python!**



What are the other alternatives for executing python?

2. Basic Syntax, Printing Statements, Indentation Rules

Let's get started with our Hello Code…

Python Basic Syntax

How to find the current version of the Python?

! before any command represent that the line is a linux command

```
!python --version
# OUTPUT : Python 3.7.12
```

2.1 Printing Statements:

Let's try printing a string that is passed onto it in a new line. Here it is 'Hello Python '.

```
print('Hello Python')
# OUTPUT : Hello Python
```

The **print()** function in Python accepts python data such as int and strings and prints them to standard output.

2.2 Indentation Rules:

- Indentation refers to the alignment spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

```
print(1)
print(5)
 print(7)
```

```
# OUTPUT :
File "<ipython-input-32-6bd85f124522>", line 3
    print(7)
    ^
IndentationError: unexpected indent
```

*Consider if condition in the below example to experience indentation, If you dont know conditions , its covered in later modules

```
# Example for an "if" condition indentation
if 5 > 2:
  print("Five is greater than two!")
#space before the start of the previous line is called indentation
#prints spring value in print function

#OUTPUT: Five is greater than two!
```

2.3 Comments :

Comments:

- Python ignores Comments while execution of code.
- Comments can be used to explain code and more readable Python code.
- Comments can be used to prevent execution when testing code.
- Single Line Comment start with #
- Multiline docstring starts with(“ “ “) three quotes (single/double) and ends with (” ” ”) three quotes.

Example:

```
# is used for single-line comment in Python(will not print in output)
```

```
# I am a Single Line comment
```

Note: ‘Ctrl + /’ is the shortcut for commenting.

Docstring:

Python docstrings are used to document the code as multi-line comment.

```
print(2+3)
''' 2+3
is 5'''
```

```
""" this is
used for
multi-line docstring """  #will print in output for documentation

#(or)

''' this is
used for
multi-line docstring '''
```

3. Variables - Identifiers

Variables or Identifiers are containers for storing data values.

Variables are example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

Valid Variables

1. Can use all Lowercase letters(a - z)
2. Can use all Uppercase letters(A - Z)
3. Can use all Numbers (Except at the first character)
4. Any character of the variable can use underscore **(_)**

Invalid Variables

- Never use any of the Symbols other than "_"
- Never use White spaces
- Never start first character with a number
- Never use Keywords/Reserved words

- Few Examples of valid identifiers:
    - **a123,**

- **_n, n_9,**
- **All_is_well**
- **aB**

- Few Examples of invalid identifiers:
  - **22**
  - $hey
  - My name

Examples:

Please modify the below variables with valid and invalid syntax to check the answers and errors.

Example:

```
x = 5
y = "John"
print(x)
# OUTPUT :  5
print(y)
# OUTPUT : John
```

3.1 Redeclaration of variables:

Python allows redeclaring the existing declared variables, It's equivalent to updating the value.

Note: The most recently declared variable will be the latest value of that variable.

Example:

```
#assigning a value
x = 5
#reassigning the value to variable
X = 6
print(x) # Latest redeclared/updated value of x is considered
# OUTPUT : 6
```

Variables are **case sensitive**

For example, **myname** and **MyName** are not the same.

Example:

```
alist = 5
Alist = 6
print(alist, Alist) # Both are different variables
# OUTPUT : 5 6
```

- Few Examples of Cases:
    - **Title Case -** HeyPython
    - **CamelCase -** heyPython
    - **SnakeCase -** hey_python_hey

```
myname = 'python' # lowercase
MYNAME = 'python' # Uppercase
Myname = 'python' # Capitalcase
myName = 'python' # Camelcase
MyName = 'python' # Titlecase
My_name = 'python' # Snakecase
my_Name = 'python' # Snakecase
```

3.2 id function:

- **id()** is a Python built-in function that returns a unique identifier for an object.
- This object's identification must be unique and consistent. As it can be seen, the function only takes one parameter and returns the object's identity.
- The id is assigned to the object when it is declared.

```
a = 90
b = 50
print(id(a))
print(id(b))
#94665375089952
#94665375088672
```

Multiple Variable Assignment:

Python provides you to assign multiple variables at a time like below -

Example:

```
a1, b2, c3 = "Hey", "Hello", "Hola"
print(a1)
# OUTPUT : Hey
print(b2)
# OUTPUT : Hello
print(c3)
# OUTPUT : Hola
```

```
a,b = 40,50
print(a,b)
print(b,a)
print(a)
print(b)
# print(a),print(b)
print(a,',',b)
#40 50
#50 40
#40
#50
#40 , 50
```

Example:

```
#assigns 'Hey' value to all the variables
a = b = c = "Hey"
print(a)
# OUTPUT : Hey
print(b)
# OUTPUT : Hey
print(c)
# OUTPUT : Hey
```


Packing - Unpacking

Declaring multiple values in one variable called as **Packing.**

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**.

Note: Concepts of List, Tuple etc., explained in later sections.

Example:

```
things = ["pencil", "pen", "sketch"]
x, y, z = things
print(x)
# OUTPUT : pencil
print(y)
# OUTPUT : pen
print(z)
# OUTPUT :sketch
```

In the same way, How to remove/delete the variable permanently from the storage?

Use the **del** keyword for it.

Example:

```
x = 5
del x
print(x) #This should raise an error that x is undefined
```

Exercise:

Find which is a valid and Invalid variable?

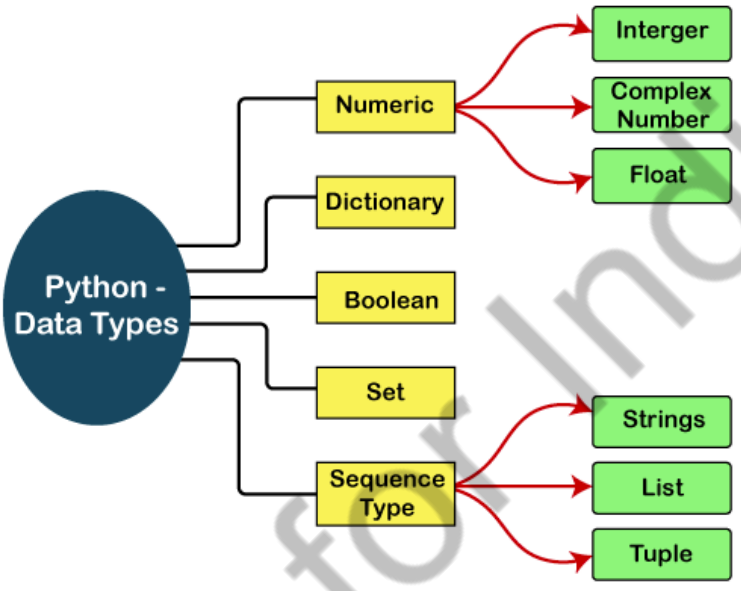
```
# abc
# a23
# &hey
# Fatima2
# Jab#neel
# _Abcv
# Py thon = 56
# ab_657 = 6
# 6ujhgft
```

5. Data Types

- In programming, the data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

String Type:	**str**

Numeric Types:	**int, float, complex**
Sequence Types:	**list, tuple**
Mapping Type:	**dict**
Set Types:	**set, frozenset**
Boolean Type:	**bool**



5.1 Datatypes:

Example:

```
# Integer
x = 20
# Alternative way of defining Integer number
x = int(20)

# Float
x = 20.5
# Alternative way of defining Float number
x = float(20.5)

# Complex
x = 1+1j
x = complex(1+1j)
# Alternative way of defining complex numbers
X = 1
```

```
Y = 1j
z = complex(X+Y)
print(z)
# OUTPUT :(1+1j)
print(z.real, z.imag)
# OUTPUT : 1.0 1.0
```

Below are examples of different other Data types. Please insert print statement and see the results-

```
# List
x = ["apple", "banana", "cherry"]
# Alternative way of defining List
x = list("apple", "banana", "cherry")

# Tuple
x = ("apple", "banana", "cherry")
# Alternative way of defining Tuple
x = tuple("apple", "banana", "cherry")

# Dictionary
x = {"name" : "John", "age" : 36}
# Alternative way of defining Dictionary
x = dict(name="John", age=36)

# Set
x = {"apple", "banana", "cherry"}
# Alternative way of defining Set
x = set("apple", "banana", "cherry")

# Frozen Set
x = frozenset({"apple", "banana", "cherry"})
# Alternative way of defining Frozen Set
x = frozenset("apple", "banana", "cherry")

# Boolean
x = True
x = False
# Alternative way of defining Boolean numbers
x = bool(5)
x = bool(0)
```

Wondering how you can check the data type of a value you declared? Well, this is how you do it.

Example:

```
x = "Hello World"
y = 20
z = -4.5
print(type(x),type(y),type(z))
# OUTPUT : <class 'str'> <class 'int'> <class 'float'>
```

range( ):

Range is also a datatype function used to specify a particular interval of values.

Syntax: **range(start_value, end_value, step_value)**

Example:

```
x = range(10)
y = range(0,10)
z = range(4,10)
print(type(x),type(y),type(z))
# OUTPUT : <class 'range'> <class 'range'> <class 'range'>
```


*Detailed Individual Datatypes discussion is done in later modules.

5.2 Type Conversions:

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

- Implicit conversion
- Explicit conversion

Implicit conversion:

Python automatically converts one data type to another data type while operating. It always converts smaller data types to larger data types(widening) to avoid the loss of data. This conversion is also called coercion.

Example-

```
x = 1    # int
y = 2.8  # float

a = x + y
b = x * y
c = y - x
d = y / x
# Final result will be in float to be accurate
print(a, type(a))
# OUTPUT :  3.8 <class 'float'>
print(b, type(b))
# OUTPUT :  2.8 <class 'float'>
print(c, type(c))
# OUTPUT : -1.7999999999999998 <class 'float'>
print(d, type(d))
# OUTPUT : 0.35714285714285715 <class 'float'>
```

In the above example **int**(smaller data type) is not able to operate with **str**(HIgher data type), In these situations we can use Explicit conversions.

Explicit conversion:

Value can be converted from one data type to another explicitly by the programmer like below-

Example:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
t = '3.2'
#convert from int to float:
a = float(x)

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

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

print(a, type(a))
```

```
# OUTPUT : 1.0 <class 'float'>
print(b, type(b))
# OUTPUT : 2 <class 'int'>
print(c, type(c))
# OUTPUT : 1+0j <class 'complex'>

print(x+int(y))
# OUTPUT : 3
print(y+float(t))
# OUTPUT : 6.0
print(x+z)
# OUTPUT : (1+1j)
```

Example:

```
x = [1,2,3,4,5]    # List
y = (1,2,3,4,5) # Tuple
z = {1,2,3,4,5}   # Set

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

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

#convert from int to complex:
c = list(z)

print(a)
# OUTPUT : (1, 2, 3, 4, 5)
print(b)
# OUTPUT : {1, 2, 3, 4, 5}
print(c)
# OUTPUT : [1, 2, 3, 4, 5]

print(type(a))
# OUTPUT : <class 'tuple'>
print(type(b))
# OUTPUT : <class 'set'>
print(type(c))
# OUTPUT : <class 'list'>
```

How can we interact with the code?

Try this for magic.

Example:

```
X = input('Enter a value:')
print(X)

'''
OUTPUT:
Enter a value:25
25
'''
```

6. Operators:

Operand , any Operator, Operand → Operation

Python has the following types of operators.

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Bitwise operators
5. Logical operators
6. Identity operators
7. Membership operators
8. Conditional (ternary) operator

6.1. Arithmetic Operators

Operator	Operation	Description
Plus (+)	x+y	Returns the addition of two operands.
Minus (-)	x-y	Returns the Subtraction of two operands.
Multiplication (*)	x*y	Returns the Product of two operands.

Division (/)	x/y	Returns the Division of two operands.
Floor Division	x//y	Returns the Floor Division of two operands.
Modulus (%)	x%y	Returns the integer remainder after dividing the two operands.
Exponentiation	x**y	Returns the first operand to the power of the second operand.

Example:

```
# Arithmetic Operators
x,y,z = 9,3,2
print(x+y) # add
print(x-y) # sub
print(x*y) # mul
print((x/y)/z) # div
print(x//y) # floor div
print(math.floor(x/y)) # rounding up to next integer value
print(math.ceil(x/y)) # rounding up to previous integer value
print(x%y) # modulus / remainder
print(x**y) # exponentiation / power

'''
OUTPUT:
12
6
27
1.5
3
3
3
0
729
'''
```

```
# print(6.1+6.2)
# print(1+2+2+432+32+32+32+3)
# print(2.1+3.2)
# print('hey'+' python')
```

```
# print([1]+[2,3])
# print((1,2)+(2,3))
# print((1)+(2,3))
# print({1,2}+{2,3})
# print({1:2}+{3:4})
```

```
# Multiplication
# print(3.1*4)

# Repetition
print('hey '*3)
print([1,2]*3)
print((1,2,3)*5)
# print({1,2}*3)
# print((1,3)*(1,23))
# print((2)*(3))
```

Exponentiation Operator Example:

```
# Exponentiation/Power operator
print(2**3) # is equal to 23
#OUTPUT : 8
```

Exponentiation operator used for finding Square root of a number-

Example:

```
var number = 625
print('Square Root of number :', number ** 0.5) # 1/2 = 0.5
#OUTPUT : 25
```

```
# exponentiation(**)
# 2*2*2*2
# 2**4
8**16+
#281474976710656
```

Remainder Example:

```
a,b = 2, 5
print(a%b) #OUTPUT : 2
'''
OUTPUT :
 5)2(2
   0
 ----
   2
So, a%b = 2
'''
```

6.2. Bitwise Operators

Operator	Usage	Description
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a zero in each bit position for which the corresponding bits of both operands are zeros.
Bitwise XOR	a ^ b	Returns a zero in each bit position for which the corresponding bits are the same. [Returns a one in each bit position for which the corresponding bits are different.]
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts a in binary representation b bits to the left, shifting in zeros from the right.
Sign-propagating right shift	a >> b	Shifts an in binary representation b bits to the right, discarding bits shifted off.

Bitwise AND assignment:

Example:

```
a,b = 2,5
print(a&b) #0
'''
 a = 2 = 0010
 b = 5 = 0101
         ----
   a&b = 0000  (Using AND truth table)
'''
```

Bitwise XOR assignment:

Example:

```
a,b = 3,5
print(a^b) #OUTPUT : 6
'''
 a = 3 = 0011
 b = 5 = 0101
    ----
   a^b = 0110 = 6 (Using XOR truth table)
'''
```

Bitwise OR assignment:

Example:

```
a, b = 2,5
print(a|b) #OUTPUT : 7
'''
 a = 2 = 0010
 b = 5 = 0101
    ----
   a|b = 0111  (Using OR truth table)
'''
```

Examples:

```
# & , |, <<, >>, ^, ~
print(12 & 10) # 8
print(12 | 10) # 14
```

```
print(20 << 3) # 160
print(20 >> 3) # 2
print(20 ^ 3)
```

```
# print(12 & 10 & 8)
print(12 | 10 | 8)
# # print(20 << 3 << 1)
# print(~7) # ~N = -(N+1)
# #  0 - POSITIVE ; 1 - Negative ; 1 --> 0; 0 --> 1
# # 3 = 0011
# # -4 = 1100
# # 7 = 00111
# # -8 = 11000
```

BITWISE NOT-

~(N) = -(N+1)

~(+N) = If N is positive, Invert bits first and then do 2's compliment on inverted output because sign should also be changed.

~(-N) = If N is Negative, Do 2's compliment to find negative binary and then invert output bits.

Ex: ~(13) ?
--> With formula
-(13+1) = -14

--> Without formula
+13 = 1101

~ is 1's = 0010 (invert bits first)

(then apply 2's compliment = 1's compliment + 1)
0000 0010
1's 1111 1101
+1 = 0000 0001

- 1110 = -14

Ex: ~(-13) ?
--> With formula
-(-13+1) = -(-12) = 12

--> Without formula
13 = 1101

(First apply 2's compliment = 1's compliment + 1)

+13 = 0000 1101
1's = 1111 0010
+1 = 0000 0001

1111 0011 (invert these bits now)

0000 1100 = +12

Left Shift:

Example:

```
num = 25
print(num << 2) #100

''' Here 25 in binary is 11001
The left shift goes on shifting the left side by appending "0" to
its right.
It gives 1100100 which is 100 in decimal
'''
```

Right Shift

Example:

```
num = 25
print(num >> 2) #6

'''
Here 25 in binary is 11001
The right shift goes on shifting the right side by eliminating the
```

```
binary values which are to their right.
It gives 110 which is 6 in decimal
'''
```

6.3. Comparison Operators:

Operator	Description
Equal (==)	Returns true if the operands are equal.
Not equal (!=)	Returns true if the operands are not equal.
Greater than (>)	Returns true if the left operand is greater than the right operand.
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.
Less than (<)	Returns true if the left operand is less than the right operand.
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.

Example:

```
# Comparison Operators
p = 399
s = 399
print(p>s) # Greater than
print(p<s) # Less than
print(p == s) # Equal to
print(p != s) # Not equal to
print(p >= s) # Greater than or equal to
print(p <= s) # Lesser than or equal to
'''
OUTPUT :
False
False
True
False
True
True
```

```
'''
```

Examples

```
# ==, !=, > , >=, < , <=
# print(2 == 2.0)
# print((2) == 2.0)
# print('2' == 2)
# print(2.00 == 2.000)
# print([1,2] == [1,2,3])
# print(7+0j == (1))
# print(7+3j == 7+3j)
# print((1,2)==(1,2))
# print({1,2} != {2,1})
# print({1,2} == {1,2})
# print({1,2} == {2,1,3})
# print({1.0,2} != {2,1})
# print(type(7/2)==type(7//2))
# print((4%2) >= (4//2))
# print((7+1j) <= (7.0+1.0j))
```

6.4. Assignment operators:

Name	Shorthand operator	Expression
Assignment	x = y	x = y
Addition assignment	x += y	x = x + y
Subtraction assignment	x -= y	x = x - y
Multiplication assignment	x *= y	x = x * y
Division assignment	x /= y	x = x / y
Floor Division assignment	x //= y	x = x // y
Remainder assignment	x %= y	x = x % y
Exponentiation assignment	x **= y	x = x ** y
Left shift assignment	x <<= y	x = x << y
Right shift assignment	x >>= y	x = x >> y
Bitwise AND assignment	x &= y	x = x & y

Bitwise XOR assignment	x ^= y	x = x ^ y
Bitwise OR assignment	x |= y	x = x | y

Examples-

```
a = 3
a+=3 #a = a+3
print(a)
```

```
a,b=2,3
a+=b # a = a + b; 2+3
print(b,a)
```

```
b,a,c = 4,5,6
b-=c+a #b=b-(c+a)
c+=b-a #c=c+(b-a)
a+=b-c #a = a+(b-c)
print(c,a,b)
```

```
a,c,b = 4,6,8
a*=b+c # a = a*(b+c) ; 4*(8+6);56
c/=b-a # c = c / (b-a) ; 6/(8-56) ; -0.125
a-=b*c # a = a-(b*c) ; 56 - (8*-0.125) = 56+1 = 57
print(a+b+c) #57+8+(-0.125)
```

```
a,c,b = 7,2,9
a <<=c #a =a<<c; 7<<2 = 28
b^=a+a # b = 9^56 ; 001001 ^ 111000 = 110001 = 49
c|=b&a # c = c | (        49 & 28; 110001 & 011100 = 010000 =16; 10 | 10000
= 10010 = 18
print(b>>(a|c)) # 49>>(28|18) ; 11100 | 10010 = 11110 = 0
```

6.5. Logical Operators

Operator	Description
Logical AND	Returns expr1 if it can be converted to false; Thus, when used with Boolean values, returns true if both operands are true; otherwise, returns false.
Logical OR	Returns expr1 if it can be converted to true; Thus, when used with Boolean values, returns true if either operand is true; if both are false, returns false.
Logical NOT	Returns false if its single operand can be converted to true; otherwise, returns true.

Example:

```
# Logical Operators
p = False
s = True
print(p and s) #Logical AND
print(p or s) # Logical OR
print(not s) # Logical NOT
'''
OUTPUT :
False
True
False
'''
```

Logical and:
1. Find first false
2. When both are true find rightmost

Truth Table for Logical AND is below:

operand1	operand2	result
false1	false2	false1
false	true	false
true	false	false
true1	true2	true2

```
# print(1 and 2)
# print(3 and 4)
# print(0 and False)
# print(0 and 2)
# print('s' and False)
# print(3 and 999)
# print('0' and False)
# print([] and ())
print('1' and 0)
```

Logical or:
1. Find first true
2. When both are false find rightmost
Truth Table for Logical OR is below:

operand1	operand2	result
false1	false2	false2
false	true	true
true	false	true
true1	true2	true1

```
print(7 or 64) # 1 or 1 = 1
print(0 or 64) # 0 or 1 = 1
print(False or 0)  # 0 or 0 = 0
print(1 or 0) # 1 or 0 = 1
print('1' or '0')
print([1,2] or (0.0))
print(0+0j or 0j)
print(0j or 0j+0)
```

Precedence order when combined :
and > or

```
# print( 6 and 7 and 0 and 80)
# print( 0 or 2 or 6)

# PREFERENCE ORDER - and, or

# print(3 and 5 or 7 or 15 and 80)
```

```
print(3 or 5 or 7 and 15 and 67 and 56 or 'hey')

# print(3 or 5 or 56 or 'hey')
```

```
# not
# print(not False)
# print(bool('hey'))
# print(not 'hey')
print(not [])

# print(not (0))
print(not (0,2))
```

6.6. Identity Operators:

Operator	Operation	Description
is	x is y	Returns True if both operands are the same object
is not	x is not y	Returns True if both operands are not the same object

Example:

```
# Logical is
p,s = 2,2
x,y = [1,2],[1,2]

print(p is s) # returns True

print(x is y) # returns False
''' returns False surprisingly? It's because the x is not the same
object as y, even if they have the same content '''


print(p == s) # returns True
'''to demonstrate the difference between "is" and "==": this
comparison returns True because x is equal to y'''
```

Example:

```
# Logical is not
p,s = 2,2
x,y = [1,2],[1,2]
Refer for reasoning: https://www.journaldev.com/22925/python-id
print(p is not s) # returns False

print(x is not y) # returns True
''' returns True surprisingly? It's because the x is not the same
object as y, even if they have the same content '''

print(p != s) # returns False
'''to demonstrate the difference between "is" and "!=": this
comparison returns True because x is not equal to y'''
```

```
# is, is not

a = {2:1}
b = {2:1}
print(id(a))
print(id(b))
print((1,2) is (1,2))
print(0.0 is 0.0)
print('a' is 'a')
print(a is not b)
print({1,2} is {1,2})
```

```
print(2 is 2)
a = 2
b = 2
print(id(a))
print(id(b))
print(a is b)
```

```
print(2.0 is 2.0)
a = 2.0
b = 2.0
print(id(a))
```

```
print(id(b))
print(a is b)
```

```
print('123' is '123')
a = '123'
b = '123'
print(id(a))
print(id(b))
print(a is b)
```

```
print((1,2) is (1,2))
a = (1,2)
b = (1,2)
print(id(a))
print(id(b))
print(a is b)
```

```
print([1,2] is [1,2])
a =  [1,2]
b =  [1,2]
print(id(a))
print(id(b))
print(a is b)
```

```
False is False
True is True
True is False
```

6.7. Membership Operators:

Operator	Operation	Description
in	x in y	Returns True if a sequence with the specified value is present in the object
not in	x not in y	Returns True if a sequence with the specified value is not present in

		the object

Example:

```
a = ['pen','book']
print("pencil" in a)
print("apple" not in a)
'''OUTPUT:
False
True
'''
```

```
# in, not in
a = [1,2,3]
# print(2 in a)
# print(2.0 in a)
print((3) not in a)
```

6.8. "Conditional Ternary" Operator:

Example:

```
# Python program to demonstrate ternary operator
a,b = 4,2

# Use tuple for selecting an item
# (if_test_false,if_test_true)[test]
print( (b, a) [a < b] )

'''Conditional territory operator returns the value 'a' if it is
true, and returns the value 'b' if it false.'''

# Use Dictionary for selecting an item
print({True: a, False: b} [a < b])
# a and b boolean values can be interchanged
```

```
# Conditional Operartor
a, b = 2,4
# a,c,d = 5,5,5
# a = c = d = 5
```

```
# print((false case, true case)[condition])
# print(('greater than','lesser than')[a>b])
```

7. Strings

- In Python, **Strings** are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a character is simply a string with a length of 1.
- Strings in python are surrounded by either single quotation marks, or double quotation marks or even triple quotation marks(multiline comments).
- Square brackets can be used to access each element of the string.

Example:

'Python' is the same as "python".

Example:

```
print("Python")  #OUTPUT : Python
print('Python')  #OUTPUT : Python

# Single quotes and double quotes Both are same for strings
```

Examples of different strings-

Example:

```
a = "This is single line string"
print(a)
#OUTPUT : This is single line string

b = """This is an
example of demonstrating you
multi-line string with double quotes"""
print(b)

#OUTPUT :
'''This is an
example of demonstrating you
multi-line string with double quotes
'''
```

```
c = '''This is an another
example of demonstrating you
multi-line string with Triple quotes.
'''
print(c)
#OUTPUT :
'''
This is an another
example of demonstrating you
multi-line string with Triple quotes.
'''
```

7.1 String Indexing:

Square brackets can be used to access elements of the string.

Note1: The first position always starts from zero (0) from left to right.

Note2: The first position always starts from minus 1 (-1) from right to left.

Note3: Even empty space in a string is one character and counted.

Example:

```
a = "Hello, Python!"
print(a[0])
print(a[1])
print(a[10])
print(a[-1])
print(a[-3])
'''
OUTPUT:
H
e
h
!
o
'''
```

Length of a string:

The **len( )** function returns the length of a string i.e., the number of characters in a string.

Example:

```
a = "Hello, Python!"
print(len(a))
#OUTPUT: 14
```

Let us apply **membership operators** to strings.

Example:

```
mystr = "You are never older to learn things"
print("learn" in mystr)
print('elder' not in mystr)

'''OUTPUT:
True
True
'''
```

7.2 Slicing Strings:

You can return a slice of a string by mentioning the range of indices separated by a colon.

Note: [Start index : End index] always returns values excluding the end index value.

Example:

```
a = "Hello, Python!"
print(a[3:8]) # excludes 8th index value
#OUTPUT : lo, P
```

Slice From the Start:

Slicing happens from the zeroth position to the given position.

Get the characters from the start to position 5 (not included):

Example:

```
a = "Hello, Python!"
print(a[:5])
#OUTPUT : Hello
```

Slice To the End:

Slicing happens from the specified position to the end:

- STRING_NAME[start_value : end_value+1 : Stepsize_value]
- end_value is always excluded
- If start_value and end_value are not mentioned, it means all.
- If only start_value is mentioned, values are from that index value to the end.
- If only end_value is mentioned, values are from starting to till the end index value.

Get the characters from position 2, and all the way to the end:

Example:

```
b = "Hello, Python!"
print(b[2:])
#OUTPUT : llo, Python!
```

7.3 How to find the position of a specific value?

Index and Find methods are used for finding the position of a specific value or specific substring.

string.index(value, start, end) **(or)**

string.find(value, start, end)

Example:

```
Alist = "Hello, python!"
A = Alist.index("p")
print(A)
#OUTPUT : 7

# Now let's check in the range 0 to 5
B = Alist.index("p",0,5) # Will raise an error - not found
print(B)
#OUTPUT :  nothing and gives Value Error

c = Alist.find("p",0,5) # Will give -1 for not found
print(c)
#OUTPUT : -1
```

Note:

- Index method gives error if an element is not found in the range
- Find method shows **-1** if an element is not found in the range
- Start and End values are optional, the default start value is 0 and the default end value is till the end.

Negative Indexing:

Use negative indices to slice from the string:

Example:

```
b = "Hello, python!"
print(b[-5:-2])
#OUTPUT : tho
```

Negative indices can also be used to slice the string in reverse order of the string where the Step value should be a negative value.

1. if start_value < end_value → Positive step_value → left to right
2. if end_value < start_value → Negative step_value → right to left

Example:

```
b = "Hello, python!"
print(b[-2:-8:-1])
#OUTPUT: nohtyp
```

All indices values:

Use just colon to get all the values from start to the end of the string:

Example:

```
b = "Hello, python!"
print(b[:])
#OUTPUT : Hello, python!
```

7.4 String Methods -

Python has a number of integrated methods, which can be used on strings.

Upper Case:

The **upper( )** method returns the parameter string in the upper case:

Example:

```
a = "Hello, python!"
print(a.upper())
#OUTPUT : HELLO, PYTHON!
```

Lower Case:

The **lower( )** method returns the string in the lower case:

Example:

```
a = "Hello, python!"
print(a.lower())
#OUTPUT :  hello, python!
```

Title Case:

The **title( )** method returns a string where the first character in each word is upper case. Like a header, or a title.

If the word contains a number or a symbol, the first letter after that will be converted to uppercase.

Example:

```
txt = "Welcome to the python world"
print(txt.title())
#OUTPUT :  Welcome To The Python World
```

Whitespace Removal:

Whitespace is the space character before and/or after the actual text.

The **strip( )** method removes whitespaces at the beginning and end of the strings only if there are.

Note: Strip method will not remove or affect in between characters or white spaces.

Example:

```
a = "   Hello, Python!   "
print(a.strip()) #removes whitespace characters in the start and
end.
#OUTPUT : Hello, Python!
```

String Count( ):

The **count( )** method returns the number of times the value appears in the string.

Example:

```
txt = "I love coding, coding is my favourite thing"
print(txt.count("coding"))
#OUTPUT :  2
```

String Replace:

The **replace( )** method replaces a particular string with a specified string.

Example:

```
a = "Hello, Python!"
print(a.replace("Hello", "Hey")) # single word replace
print(a.replace("Hello", "Hey")).replace("Python","c")#multiple replace
'''
OUTPUT :
Hello, Python!
Hey, Python!
Hey, c
'''
```

Split String:

Split method is used to split the string into multiple parts where each of the parts should be separated/delimited by a given character. Such a character will be passed as a parameter to the split method.

The **split( )** splits the string into substrings.

Example:

```
a = "Hello, Python!"
print(a.split()) #default will split into words
print(a.split(",")) #specific character splitting
'''
OUTPUT :
['Hello,', 'Python!']
['Hello', ' Python!']
'''
```

Join String:

The join( ) method takes all items in an iterable and joins them into one string and all the items will be separated by a special character in a string.

A string must be specified as the separator.

Example:

```
Mylist = ['i','am','learning','python']
x = " ".join(Mylist)
Y = "$".join(Mylist)
print(x)
print(Y)

'''
OUTPUT:
i am learning python
i$am$learning$python
'''
```

There are few more predefined methods that can be applied to strings. You can find them here- W3Schools Python String Methods

String Concatenation

To concatenate, or combine, two strings you can use the +(plus) operator.

Example:

```
a = "Hello"
b = "python"
c = a + b
print(c)
#OUTPUT : Hellopython
```

You can also use + for adding spaces in between strings.

Example:

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
#OUTPUT :  Hello World
```

How to reverse a string?

Before we discuss reversing, we need to know about the step size parameter in strings.

string([start_value : End_value : step_size])

Example:

```
string = 'akhil'
print(string[::-1]) # reverse the string
print(string[::]) # by default step size is 1
print(string[::2]) # step size 2
'''
OUTPUT :
lihka
akhil
Ahl
'''
```

Python String Formats:

Note: We cannot join a number to a string using the + operator. Give it a try.

Hence we come up with string formats.

The **format()** method takes an unlimited number of arguments, and are placed into the respective placeholders:

```
age = 30
cars = 6
years = 10
myorder = "At the age of {}, I had {} cars which i used for {}
years."
print(myorder.format(age, cars, years))
```

```
#OUTPUT :  At the age of 30, I had 6 cars which i used for 10
years.
```

In the above example, the input is taken as per the order given in the print statement.

We can modify the order using index values like below.

Example:

```
age = 30
cars = 6
years = 10
myorder = "At the age of {2}, I had {0} cars which i used for {1}
years."
print(myorder.format(cars, years, age))
#OUTPUT :  At the age of 30, I had 6 cars which I used for 10 years.
```

Escape Character:

Escape characters are used to avoid the clash with string representations.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

Example:

```
txt = "We are the so-called "Vikings" from the north." # Wrong
txt = "We are the so-called \"Vikings\" from the north." # Correct
print(txt)
#OUTPUT :  We are the so-called "Vikings" from the north
```

Please refer to the below link for more string methods -

https://docs.python.org/3/library/stdtypes.html#string-methods

8. Booleans:

Boolean values are: **True** or **False**.

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

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

Often valuewise:

- True refers to 1 or Success case
- False refers to 0 or failure case

Example:

```
print(5 > 4) # True because 5 is greater than 4
print(5 == 8) # False because 5 is not equal to 8
print(7 >= 3) # True because 7 is greater than equal to 3
print(11 < 9) # False because 9 is not less than 9
'''
OUTPUT :
True
False
True
False
'''
# Below all are few examples for Boolean True because there is
some value
print(bool("Hello")) #OUTPUT : True
print(bool(15)) #OUTPUT : True
print(bool("abc"))  #OUTPUT : True
print(bool(123))  #OUTPUT : True
print(bool(["apple", "cherry", "banana"]))  #OUTPUT : True

# Below all are few examples for Boolean False because they are
null values
bool(None)  #OUTPUT : False
bool(0)   #OUTPUT : False
bool("")   #OUTPUT : False
bool(())   #OUTPUT : False
```


9. List:

- Python list is one of the 4 built-in data types used to store multiple items in a single variable.
- The other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
- A List is a collection of heterogeneous items or elements that allow modifications.
- A List is created using square brackets with elements enclosed in it.

Example:

```
Alist1 = ["one", "two", "three"]
Alist2 = list(("one", "two", "three")) # Alternative creation of list
print(Alist1)
print(Alist2)
'''
OUTPUT :
['one', 'two', 'three']
['one', 'two', 'three']
'''
```

9.1 Key Points:

- Lists can have duplicate values.
- Lists are Ordered with specific positions(index values).
- Lists are changeable(add, remove, update).
- Lists allow all the data type values as an item.

Example:

```
# Single Data Type items List
Alist1 = ["one", "two", "three"]
Alist2 = [1,2, 3]
Alist3 = [True, False, True]
print(Alist1,Alist2,Alist3)
#OUTPUT :  ['one', 'two', 'three'] [1, 2, 3] [True, False, True]

# Multiple data type items list
Alist4 = ["one", 2, True, ['two','three']]
print(Alist4)
'''
OUTPUT:
['one', 'two', 'three'] [1, 2, 3] [True, False, True]
['one', 2, True, ['two', 'three']]
'''
```

List Length:

The length of the list is nothing but the number of items in the list. Use **len( )** to get it.

Example:

```
Alist = ["one", "two", "three", "two", "Two"]
print(len(Alist))
#OUTPUT :  5
```

Lists allow duplicate values-

Example:

```
Alist = ["one", "two", "three", "two", "Two"]
print(Alist)
#OUTPUT :  ['one', 'two', 'three', 'two', 'Two']
# Note - "Two" is different from "two", hence not a duplicate of "two"
```

A List is Ordered with specific positions(index values) -

Note1: The first position always starts from zero(0) from left to right.

Note2: The first position always starts from minus one(-1) from right to left.

Note3: The end value in the range(i.e., the immediate value after the colon) is always excluded.

Example:

```
Alist = ["one", "two", "three", "four", "five"]
# Index value accesing is same as the string here too
print(Alist[0]) # First position from left to right
print(Alist[-1]) # First position from right to left
print(Alist[2:4]) # Range from 2-4(excluded)
print(Alist[-3:-1]) # Range from -3 to -1(excluded)
print(Alist[:4]) # Range from the start till 4(excluded)
print(Alist[2:])# Range from 2 till the end
print(Alist[:]) # All the values
'''
OUTPUT :
one
five
['three', 'four']
['three', 'four']
['one', 'two', 'three', 'four']
['three', 'four', 'five']
['one', 'two', 'three', 'four', 'five']
```

```
'''
```

How to find the position of a specific value?

Example:

```
Alist = ["one", "two", "three", "two", "Two"]
A = Alist.index("three")
print(A)
#OUTPUT : 2
```

Note: There is no find( ) method for lists like Strings to find the position.

9.2 List Methods:

Lists are mutable(add, remove, update) -

Update a single item -

Example:

```
Alist = ["one", "two", "three", "two", "Two"]
Alist[3] = "Four"
print(Alist)
#OUTPUT : ['one', 'two', 'three', 'Four', 'Two']
```

Update multiple items -

Example:

```
Alist = ["one", "two", "three", "two", "Two"]
Alist[3:] = ["Four", "Five"]
print(Alist)
#OUTPUT : ['one', 'two', 'three', 'Four', 'Five']
```

Example:

```
Alist = ["one", "two", "three", "two", "Two"]
Alist[3] = ["Four", "Five"]
print(Alist) # Updates the item in that position as a sub-list
#OUTPUT : ['one', 'two', 'three', ['Four', 'Five'], 'Two']
```

Example:

```
Alist = ["one", "two", "three", "two", "Two"]
Alist[1:3] = ["Four", "Five"]
print(Alist) # Updates the items in those respective positions
#OUTPUT : ['one', 'Four', 'Five', 'two', 'Two']
```

Example:

```
Alist = ["one", "two", "three", "two"]
Alist[1:3] = ["Four"]
print(Alist) # Updates the items in those respective positions
#OUTPUT :  ['one', 'Four', 'two']
```

All the above examples showed us how to replace/update the values, We will now see how to add and insert the items to the list.

append( ):

This method will append values only to the end.

Example:

```
Alist = ["one", "two", "three", "four"]
Alist.append("five") # It appends item only at the last
print(Alist)
#OUTPUT : ['one', 'two', 'three', 'four', 'five']
```

insert( ):

Insert method is used to insert items in a specified position without replacing the items.

Example:

```
Alist = ["one", "two", "three", "five"]
Alist.insert(3,"Four")
print(Alist)
#OUTPUT :  ['one', 'two', 'three', 'Four', 'five']
```

extend( ):

To append elements from another list to the current list, use the extend( ) method.

Example:

```
Alist1 = ["one", "two", "three"]
Alist2 = [4,5,6]
```

```
Alist1.extend(Alist2)
print(Alist1)
#OUTPUT :  ['one', 'two', 'three', 4, 5, 6]
```

Note: In this case, elements will be added at the end only.

Alternative way of doing this is using '+' in between lists:

Example:

```
Alist1 = ["one", "two", "three"]
Alist2 = [4,5,6]
Blist = Alist1 + Alist2
print(Blist)
#OUTPUT :  ['one', 'two', 'three', 4, 5, 6]
```

Eventually, Extend method, Append method and plus operator does the same operation.

remove( ):

Remove method helps us to remove items from the list.

Example:

```
Alist = ["one", "two", "three", "five"]
Alist.remove("three")
print(Alist)
#OUTPUT : ['one', 'two', 'five']
```

pop( ):

The pop method helps us to remove specific index items from the list.

Example:

```
Alist = ["one", "two", "three", "five"]
Alist.pop(2) # Here 2 is the index value
Alist.pop() # if you don't pass any index value, last item is removed
print(Alist)
#OUTPUT :  ['one', 'two']
```

del Keyword:

An alternative to the pop method to remove specific position values is using keyword **del**.

Example:

```
Alist = ["one", "two", "three", "five"]
del Alist[1] # Will remove the value at index 1
print(Alist)
del Alist # Will remove entire variable
#OUTPUT : ['one', 'three', 'five']
```

If there is a case where you want all the values of the variable to be removed but not the variable itself. That is where you can use the **clear( )** method.

clear( ):

The **clear( )** method empties the list.

The list still remains, but it has no content.

Example:

```
Alist = ["one", "two", "three", "five"]
Alist.clear() # Will remove all the values but not variable
print(Alist)
#OUTPUT : []
```

sort( ):

If you want to arrange your values in a particular order, the **sort( )** method is handy.

Note: By default, the sort method does ascend order.

Example:

```
Alist = ["one", "two", "three", "five"]
Alist.sort() # Will sort all the values alphabetically ascending order
print(Alist)
#OUTPUT : ['five', 'one', 'three', 'two']
```

Example:

```
Alist = [4,3,1,2,5]
Alist.sort() # Will sort all the values numerically ascending order
print(Alist)
#OUTPUT :  [1, 2, 3, 4, 5]
```

How to put it in descending order?

Example:

```
Alist = [4,3,1,2,5]
Alist.sort(reverse = True) # Will sort numerically  descending order
print(Alist)
#OUTPUT :  [5, 4, 3, 2, 1]
```

Note: The arrangement is purely dependent on ASCII Values. The order goes like this. Ascending Order - Numbers, Uppercase, Lowercase and Vice-versa for Descending order.

Wait, How do I reverse the elements as it is irrespective of ascending or descending order, Well, we have a **reverse( )** method.

Example:

```
Alist = [4,3,1,2,5]
Alist.reverse() # Will sort all the values in reverse order
print(Alist)
#OUTPUT :  [5, 2, 1, 3, 4]
```

How do I make a copy of a list?

Unlike Variables, you cannot do list1 = list2 because list2 acts as a reference for list1 and any changes you make list2 will reflect in list1 too. If you don't want this to happen.

There are two ways to make it happen-

Example:

```
Alist = [4,3,1,2,5]
Blist = Alist.copy() # Will sort all the values in reverse order
print(Blist)
#OUTPUT : [4, 3, 1, 2, 5]
```

Another way to make a copy is to use the built-in method **list( )**.

Example:

```
Alist = [4,3,1,2,5]
Blist = list(Alist) # Will copy all the values from a list
print(Blist)
#OUTPUT : [4, 3, 1, 2, 5]
```

9.3 List Unpacking:

When we create a list, we normally assign values to it. This is called "packing" a list:

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking".

Example:

```
fruits = ["apple", "banana", "cherry", "strawberry", "raspberry"] #pack
[green, yellow, red, pink, orange] = fruits #Unpacking
print(green, yellow, red, pink, orange)
#OUTPUT : apple banana cherry strawberry raspberry
```

***Asterisk**

If the number of elements in the list is not known in prior, you can add an * to the variable name and the elements will be assigned to that variable as a list:

Example:

```
fruits = ["apple", "banana", "cherry", "strawberry", "raspberry"]
[green, yellow, *red] = fruits
print(green)
print(yellow)
print(red)
'''
OUTPUT:
apple
banana
['cherry', 'strawberry', 'raspberry']
'''
```

Multi Dimensional List element accessing example:

```
a = ['a','b',['c','d'],'e']
print(len(a))
print(a[2:3])
print(a[2:3][0])
print(a[2:3][0][0])
```

9.4 Few more Important Functions

Sorting the List:

Please refer ASCII table while sorting

Resource: https://computersciencewiki.org/images/3/3d/Ascii_table.png

```
a = ['a','A','e','f','C','c','b']
a.sort()
```

```
a
```

```
a = [3,4,5,8,12,33,9,343,9,39]
a.sort()
a
```

```
a = ['ab','bd','acb','bcda']
a.sort(key=len)
a
```

min()

Returns minimum item among the list

```
a = [3,4,5,8,12,33,9,343,9,39]
a.min()
a
```

max()

Returns maximum item among the list

```
a = [3,4,5,8,12,33,9,343,9,39]
a.max()
a
```

enumerate()

- The enumerate object yields pairs containing a count (from start, which defaults to zero) and a value yielded by the iterable argument.
- enumerate is useful for obtaining an indexed list:

```
a = ['a','b','c','d','e']

print(list(enumerate(a))) #gives a list of tuples having indexvalue,item
print(list(enumerate(a,10))) #starts index value from 10
```

---

10. Tuple:

Tuples are also used to store multiple items in one variable as a sequence type. Yet there are differences between a list and tuple. Let's find out why-

Example:

```
Atuple = (4,3,1,2,5)
Atuple2 = tuple((1,2, 3)) # Alternative way of declaring a tuple
print(Atuple,Atuple2)
#OUTPUT : (4, 3, 1, 2, 5) (1, 2, 3)
```

10.1 Key Points:

- A Tuple can have duplicate values.
- A tuple is Ordered with specific positions(index values).
- A tuple is unchangeable(add, remove, update).

A Tuple can contain single data type items or multiple data type items.

Example:

```
# Single data Type items Tuple
Atuple1 = ("one", "two", "three")
Atuple2 = tuple((1,2, 3)) # Alternative way of declaring a tuple
Atuple3 = (True, False, True)
print(Atuple1, Atuple2, Atuple3)
#OUTPUT : ('one', 'two', 'three') (1, 2, 3) (True, False, True)

# Multiple data type items Tuple
Atuple4 = ("one", 2, True, ['two','three'])
print(Atuple4)
#OUTPUT : ('one', 2, True, ['two', 'three'])
```


Tuple Length:

The length of the tuple is nothing but the number of items in the tuple. Use **len( )** to get it.

Example:

```
Atuple = ("one", "two", "three", "two", "Two")
print(len(Atuple))
#OUTPUT : 5
```

A Tuple is Ordered with specific positions(index values) -

Note1: The first position always starts from zero(0) from left to right.

Note2: The first position always starts from minus one(-1) from right to left.

Note3: The end value in the range(i.e., the immediate value after the colon) is always excluded.

Example:

```
Atuple = ("one", "two", "three", "four", "five")
# Index value accesing is same as the string here too
print(Atuple[0]) # First position from left to right
print(Atuple[-1]) # First position from right to left
print(Atuple[2:4]) # Range from 2-4(excluded)
print(Atuple[-3:-1]) # Range from -3 to -1(excluded)
print(Atuple[:4]) # Range from the start till 4(excluded)
print(Atuple[2:])# Range from 2 till the end
print(Atuple[:]) # All the values
'''
OUTPUT :
one
five
('three', 'four')
('three', 'four')
('one', 'two', 'three', 'four')
('three', 'four', 'five')
('one', 'two', 'three', 'four', 'five')
'''
```

How to find the position of a specific value?

Example:

```
Atuple = ("one", "two", "three", "two", "Two")
A = Atuple.index("three")
print(A)
#OUTPUT : 2
```

Note: There is no find( ) method for tuples like Strings to find the position.

10.2 Tuple is Immutable:

- Tuples are Ordered
- Tuples allows duplicates
- Tuples do not allow adding/replacing/removal of items
- Tuples are IMMUTABLE
- Tuples allows all other data types as items

Like Lists, Tuples are not changeable. You cannot Add, Remove, Update the items in the tuple.

Yet, there is a way to do it.

Convert the tuples to list using type conversion and then modify the data.

Example:

```
Atuple = ("one", "two", "three", "two", "Two")
A = list(Atuple) # Converting tuple to list for modifying data
A[0] = "ONE" # some modification
Atuple = tuple(A) # Converting list to tuple after modifying data
print(Atuple)
#OUTPUT : ('ONE', 'two', 'three', 'two', 'Two')
```

del Keyword:

By using keyword **del**, you can only remove the entire tuple but not items.

Example:

```
Atuple = ("one", "two", "three", "five")
del Atuple[1] # Unsupported! Will not remove the value at index 1
#ERROR TypeError: 'tuple' object doesn't support item deletion

print(Atuple)
#OUTPUT : ('one', 'two', 'three', 'five')

del Atuple # Will remove entire tuple
```

10.3 Tuple Unpacking:

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

But, in Python, we are also allowed to extract the values back into variables. This is called

"unpacking".

Example:

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry") #pack
(green, yellow, red, pink, orange) = fruits #Unpack
print(green, yellow, red, pink, orange)
#OUTPUT : apple banana cherry strawberry raspberry
```

***Asterisk**

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a tuple:

Example:

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
'''
OUTPUT :
apple
banana
['cherry', 'strawberry', 'raspberry']
'''
```

Join Two Tuples:

To join two or more tuples you can use the + operator:

Example:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
#OUTPUT : ('a', 'b', 'c', 1, 2, 3)
```

Repeat Tuples:

If you want to repeat the content of a tuple a given number of times, you can use the * operator:

Example:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple) # fruits tuple items will be repeated twice
#OUTPUT : ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

10.4 Few more Important functions

Sorting the Tuple:

Please refer ASCII table while sorting

Resource: https://computersciencewiki.org/images/3/3d/Ascii_table.png

```
a = ('a','A','e','f','C','c','b')
a.sort()
a
```

```
a = (3,4,5,8,12,33,9,343,9,39)
a.sort()
a
```

```
a = ('ab','bd','acb','bcda')
a.sort(key=len)
a
```

min()

Returns minimum item among the tuple

```
a = (3,4,5,8,12,33,9,343,9,39)
a.min()
a
```

max()

Returns maximum item among the tuple

```
a = (3,4,5,8,12,33,9,343,9,39)
a.max()
```

```
a
```

enumerate()

- The enumerate object yields pairs containing a count (from start, which defaults to zero) and a value yielded by the iterable argument.
- enumerate is useful for obtaining an indexed list:

```
a = ('a','b','c','d','e')

print(tuple(enumerate(a))) #gives a tuple of tuples having indexvalue,item
print(tuple(enumerate(a,10))) #starts index value from 10
```

Dictionary:

Dictionary is also used to store values in **key : value** pairs.

Example:

```
Adict1 = {"one" :"two", "three":4}
Adict2 = dict({"one" :"two", "three":4}) # Alternative creation of dict
print(Adict1, Adict2)
#OUTPUT :  {'one': 'two', 'three': 4} {'one': 'two', 'three': 4}
```

Key Points:

- Dictionaries are **ordered** and you can only access values through **keys only**, but not with index value.
- Dictionaries **keys** cannot be Mutable data types as they should be unchanged.
- Dictionaries are **changeable.**(Add, remove, change)
- Dictionaries cannot have two items with the same key, if does, it updates with latest value. So, Dictionaries **do not** allow **duplicate keys**.

Dictionaries Length:

The length of the dictionary is nothing but the number of items in the dictionary. Use **len( )** to get it.

Example:

```
Adict = {"brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]}
print(len(Adict))
#OUTPUT :  4
```

How to access items?

Example:

```
Adict = {"brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]}
print(Adict["year"])
#OUTPUT :  1964
```

Alternative way of accessing values -

get()

Doesnt throw error incase if key is not available in the dictionary.

Example:

```
Adict = {"brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]}
print(Adict.get("year")) #Using get method
#OUTPUT : 1964
```

keys( ):

The **keys( )** method will return a list of all the keys in the dictionary.

Example:

```
Adict = {"brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]}
print(Adict.keys()) # will print all keys only
#OUTPUT : dict_keys(['brand', 'electric', 'year', 'colors'])
```

values( ):

The **values( )** method will return a list of all the values in the dictionary.

Example:

```
Adict = {"brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]}
print(Adict.values()) # will print all values only
#OUTPUT : dict_values(['Ford', False, 1964, ['red', 'white',
'blue']])
```

items( ):

The **items( )** method will return each item as tuples in a list in the dictionary.

Example:

```
Adict = {"brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]}
print(Adict.items()) # will print all items in tuples list

'''
OUTPUT :
dict_items([('brand', 'Ford'), ('electric', False), ('year',
1964), ('colors', ['red', 'white', 'blue'])])
'''
```

Making a change in the original dictionary will update the dictionary as well.

Example:

```
Adict = {"brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]}
print(Adict.items()) # prints all items in tuples list before
change
Adict['year'] = 2021
print(Adict.items()) # prints all items in tuples list after
change
```

```
'''
OUTPUT :
dict_items([('brand', 'Ford'), ('electric', False), ('year',
1964), ('colors', ['red', 'white', 'blue'])])
dict_items([('brand', 'Ford'), ('electric', False), ('year',
2021), ('colors', ['red', 'white', 'blue'])])
'''
```

Just like Lists, you can add, remove, pop, update, delete the dictionary or you can clear the items of dictionary.

You can create nested dictionaries as well.

Example:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```


setdefault()

Setdefault method returns the value of the given key if exists, or it creates an item with given key,value pair.

```
a = { 0 : 10,   2 : 'asia', 3: False,  1 : 11.9}
a.setdefault(20,'africa')
a
```

fromkeys()

Fromkeys method returns a dictionary with the given keys and value as input.

```
a = (1,2,3)
b = 'hey'
# {}.fromkeys(a,b)
dict.fromkeys(a,b)
```

And few more methods which are almost same as previous datatypes.

11. Set:

Set is also used to store multiple items in one variable as a sequence type without duplication of the elements. Let's find out few more details about it.

Example:

```
Aset1 = {"one", "two", "three"}
Aset2 = set(("one", "two", "three")) # Alternative creation of
list
print(Aset1, Aset2)
#OUTPUT : {'one', 'three', 'two'} {'one', 'three', 'two'}
```

```
a = {1,2,3,4}
a = set({1,2,3,4})
type(a)
```

```
a = {0,2,3.7,'45',(1,12),True, False}
type(a)
a
```


11.1 Key Points:

- Set is **not indexed**.
- Set is **not ordered.** Set items can appear in a different order every time you use them.
- Sets are **mutable** but **unchangeable,** meaning that we cannot change the items after the set has been created but can be removed.

- Sets **do not** allow **duplicate values.**
- You cannot have lists, dictionaries as elements of a set as they are mutable datatypes.
- You cannot have a set as an item as it violates mathematical interpretation.

Example:

```
# Single data Type items set
Aset1 = {"one", "two", "three"}
Aset2 = set((1,2, 3)) # Alternative way of declaring a set
Aset3 = {True, False, True}
print(Aset1,Aset2,Aset3)

# Multiple data type items set
Aset1 = {"one", 2, True, ('two','three'), True}
print(Aset1)
#OUTPUT : {'one', 2, ('two', 'three'), True}
```

Set Length:

The length of the tuple is nothing but the number of items in the tuple. Use **len( )** to get it.

Example:

```
Aset = {"one", "two", "three", "two", "Two"}
print(len(Aset))
#OUTPUT : 4
```

```
a = {1,2,3,4,5}
b = {5,6,7,8}
# a+b error, union or add method should be used
```

add( ):

To add one item to a set use the **add( )** method.

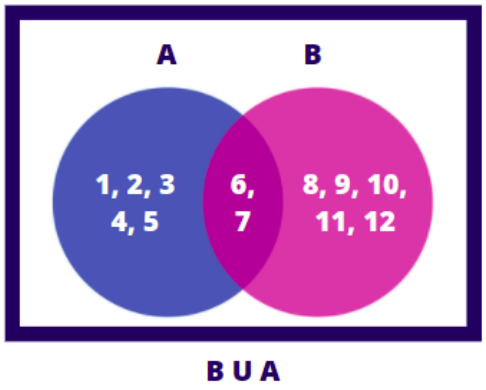
Example:

```
Aset = set({"one", "two"})
Aset.add("three")
print(Aset)
#OUTPUT : {'three', 'one', 'two'}
```

union():

Collection of all the items in both sets

Example:




```
a = {1,2,3,4,5,6,7}
b = {6,7,8,9,10,11,12}

c = a.union(b)
c
```

update( ):

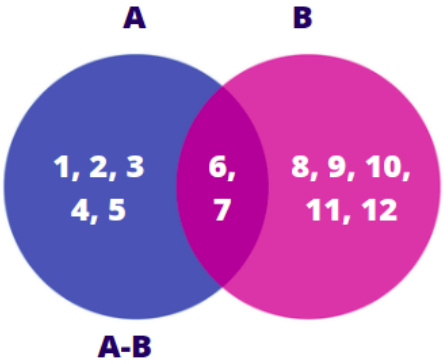
To add items from another set into the current set, you can also use the **update( )** method.

Example:

```
Aset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
Aset.update(mylist)
print(Aset)
#OUTPUT :  {'cherry', 'kiwi', 'orange', 'apple', 'banana'}
```

difference( ):

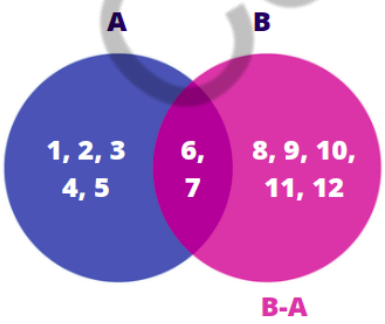
Leftover elements in first operand after removal of common elements



Example:

```
a = {1,2,3,4,5,6,7}
b = {6,7,8,9,10,11,12}

a-b

# or you can use below methods

c = a.difference(b)
c

# or you can use below method to avoid redeclaration

a.difference_update(b)
a
```

Example:
b-a



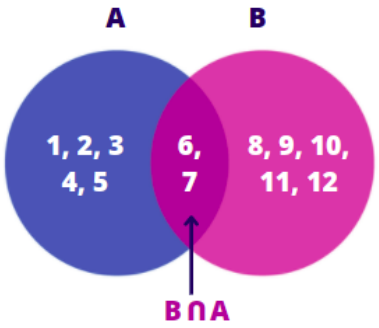

```
a = {1,2,3,4,5,6,7}
b = {6,7,8,9,10,11,12}
```

```
c = b.difference(a)
c

# or you can use below method to avoid redeclaration

b.difference_update(a)
a
```

intersection():

Common elements in both sets only

Example:




```
a = {1,2,3,4,5,6,7}
b = {6,7,8,9,10,11,12}

c = a.intersection(b)
c

# or you can use below method to avoid redeclaration

a.intersection_update(b)
a
```

disjoint():

If No element is in common in both of the sets, they are called as Disjoint sets.

Example:

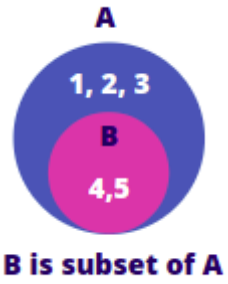
Disjoint Sets

```
a = {1,2,3,4,5}
b = {8,9,10,11,12}

c = a.isdisjoint(b)
c
```

subset():

If no element in set **b** is other than one of the elements of another set **a**, then **b** is called as subset of **b**.

Example:



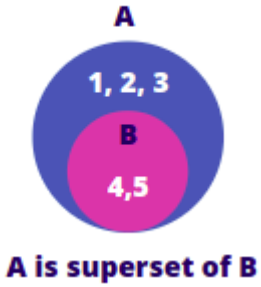
B is subset of A

```
a = {1,2,3,4,5}
b = {4,5}

c = b.issubset(a)
c
```

superset():

If all the elements in set **b** are already the elements of another set **a**, then **a** is called as superset of **b**.

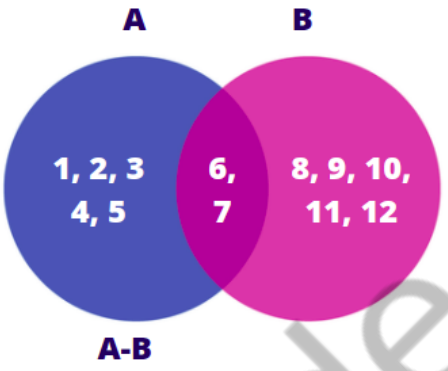
Example:




```
a = {1,2,3,4,5}
b = {4,5}

c = a.issuperset(b)
c
```

symmetric_difference( ):

Leftover elements in first operand after removal of common elements



Example:

```
a = {1,2,3,4,5,6,7}
b = {6,7,8,9,10,11,12}

a-b

# or you can use below methods

c = a.symmetric_difference(b)
c

# or you can use below method to avoid redeclaration

a.symmetric_difference_update(b)
a
```

Just like Lists, you can add, remove, pop, update, clear the sets.

Frozen Set

Unchangeable set is called as Frozen set.

```
a = frozenset({1,2,3,4})
type(a)
```

Frozenset too has mathematical set methods in similar, but it doesnt allow to change the values.

For Example:

```
a = frozenset({1,2,3,4})
a[1] = 5
#Throws an error

a.pop()
#Throws an error
```

Control flow:

1. Sequential operation
2. Decisive/Conditional operation
3. Iterative operation

1. Sequential Operation

Python is an Interpreted language, hence we know that it does line by line execution.

The execution of a program following in sequential manner is called Sequential Operation.

All the programming we have done so far is Sequential Operation.

2. Decisive/Conditional Operation:

The execution of the line depends on particular events or conditions. Such operations are called as Conditional/DecisiveOperation.

There are four ways of writing conditions based on the number of cases to be dealt.

1. if condition
2. if-else condition
3. if-elif-else condition
4. Nested condition

In Programming, there is a huge need for conditional based execution because of logical dependency.

Conditions help in doing that job.

- **Success Case** refers to **true** or **1** or having some **value**
- **Failure Case** refers to **false** or **0** or **empty**

if condition

Note:

- Any logic written inside the **if** condition is executable if and only if the condition is **True**.

Mandatory format of 'if' condition is -

- Keyword **' if '**
- **Colon** in the end of if condition
- Indentation before the start of the next line.

Syntax:

```
if (condition):
  statement/task here
```

```
if 1:
  print('hey0')
if True:
  print('hey1')
```

```
hey0
hey1
```

```
if [1,2]:
  print('hey3')
if 0:
  print('hey4')
if [0]:
  print('hey5')

hey3
hey5
```

```
if False:
  print('hey6')
if (0):
  print('hey7')
```

```
if ((0) and '[]'):
  print('hey')
if (8&7):
  print('hey')
if (4<3):
  print('hey')
if ([1,2]==[1,2]):
  print('hey1')
if ((1,2) is (1,2)):
  print('hey2')
```

Examples:

```
A = 2
B = 3
if B > A:
  print('B is greater than A')
#OUTPUT : B is greater than A
```

You can also use logical **and,** logical **or** conditions -

Example:

```
A,B,C = 3,2,1
if A>B and B>C:
  print('Both conditions must be True to get here')
#OUTPUT : Both conditions must be True to get here
```

```
A,B,C = 3,2,1
if A>B or A>C:
  print('Atleast one between both conditions must be True to get
here')
#OUTPUT : Atleast one between both conditions must be True to get
here
```

Find whether given number is positive or negative or Zero?

```
# Find whether given number is positive or negative or Zero
a = int(input('Enter a number: '))
# a = 0
if a>0:
  print ( ' a is positive number ')
if a<0:
  print ('a is negative number ')
if a==0:
  print('a is zero')


Enter a number: 3
 a is positive number
```

Find whether given number is less than 10 or less than 20 or less than 30?

```
# Find whether given number is less than 10 or less than 20 or
less than 30
# 0-10, 10-20, 20-30
a = int(input('Enter a number: '))
# a = 0
if a>=0 and a<=10:
  print("a is less than 10")
if a>=11 and a<=20:
```

```
  print('a is less than 20')
if a>=21 and a<=30:
  print('a is less than 30')

Enter a number: 3
a is less than 10
```

```
# if a number is multiple of 3  but not 5 print fizz
# if a number is multiple of 5  but not 3 print buzz
# if a number is multiple of both print fizz buzz
a=int(input('Enter a Num: '))
if ((a%3)==0 and (a%5)!=0):
   print ("Fizz")
if ((a%5)==0 and (a%3)!=0):
   print("Buzz")
if ((a%5)==0 and (a%3)==0):
   print("Fizz Buzz")

Enter a Num: 3
Fizz
```

if - else Condition

- Any logic written inside the **if** condition is executable if and only if the condition is **True**.
- Rest all cases can be executed inside the else condition in case if the condition fails.

In case if the condition fails, we can have a backup execution using the '**else**' condition.

```
A = 2
B = 1
if B > A:
  print('B is greater than A')
else:
  print('B is less than or equal to  A')
```

```
# if - else condition
a = -34
```

```
if a>0:
  print('positive number')
if a==0:
  print('equal')
else:
  print('a is negative number ')
```

```
# Find a number whether its even or odd?
a=int(input('Enter a Num: '))
if (a%2)==0:
 print('a is even')
else:
  print('a is odd')

Enter a Num: 3
a is odd
```

if - elif Condition

Note:

- Any logic written inside the **if** condition is executable if and only if the condition is **True**.
- In case if the condition fails, we can have a backup execution using the else if condition which is written as **'elif'** condition in python.
- Rest all cases can be executed inside the **else** condition in case if the previous condition also fails.

Example:

```
A = 2
B = 1
if B > A:
  print('B is greater than A')
elif B < A:
  print('B is less than A')
#OUTPUT : B Must be lesser than A
```

In case you fail in both the above conditions, **'else'** is what is left to you.

Example:

```
A = 2
B = 1
if B > A:
  print('B is greater than A')
elif B < A:
  print('B is less than A')
else:
  print('B Must be equal to A')
#OUTPUT :  B is less than A
```

Example:

```
a,b = 9,9

if  (a == b ) or (a<b): # True |  False = True
 print('I am first')

elif (a >= b) and (a<b): # True & False = False
 print('I am second')

else:
 print('I am Third')
#OUTPUT : I am first
```

if a number is multiple of 3 but not 5 print fizz
if a number is multiple of 5 but not 3 print buzz
if a number is multiple of both print fizz buzz
if a number is not a multiple of both, print not fizz buzz

```
# if a number is multiple of 3  but not 5 print fizz
# if a number is multiple of 5  but not 3 print buzz
# if a number is multiple of both print fizz buzz
# if a number is not a multiple of both, print not fizz buzz
a=int(input('enter a number= '))
if ((a%3)==0 and (a%5)!=0):
  print('fuzz')
elif ((a%5)==0 and (a%3)!=0):
  print('buzz')
elif ((a%5==0) and (a%3==0)):
  print("fizz buzz")
else:
```

```
  print('a is not divisible by both 3 and 5')

enter a number= 3
fuzz
```

Short Hand if :

To keep it in a simple way, we can write entire if condition in one line like this-

Example:

```
A = 2
B = 1
#(if condition): (if- expression)
if (A > B): print('A is greater than B')

#OUTPUT : A is greater than B
```

Example:

```
# Regular if
a = 6
if ((a%2)==0) :
 print('even')


# short hand if
if ((a%2)==0) : print('even')
```

Short Hand if...else :

To keep it in a simple way, we can write entire if...else condition in one line like this-

Note: No colon needed after the conditions

Example:

```
A = 2
B = 10
# (if- expression) (if condition) (else condition) (else expression)
print('B is greater') if (B > A) else print('A is greater')
#OUTPUT :  B is greater
```

Example:

```
a = int(input())
# Regular if-else
if ((a%2)==0) :
 print('even')
else:
 print('odd')

# (if expression) (if condition) (else) (else expression)
print('even') if ((a%2)==0) else print('odd')
```

Short Hand if-elif-else :

You can also have multiple else statements on the same line:

Note: No colon needed after the conditions.

Example:

```
A = 2
B = 1
# (if- expression) (if condition) (else condition) (elif-expression) (if
condition) else (else expression)
print('B') if B > A else print('=') if (A==B) else print('A')
#OUTPUT : A
```

Example:

```
# Shorthand if-elif-else
# (if- expression) (if condition) (else condition)
(elif-expression) (if condition) else (else expression)

a = int(input())
# Regular if-elif-else
if a>0:
 print('positive')
elif a<0:
 print('negative')
else:
 print('zero')

# shorthand if-elif-else
print('positive') if (a>0) else print('Negative') if (a<0) else
print('zero')
```

Nested conditions:

Example:

```
# -ve = -infinity to -1
# Zero = 0
# +ve = 1 to infinite

a = 5
if a >= 0:
 if a == 0:
   print('Number is zero')
 else:
   print('Its a positive number')
else:
 print('Its a Negative Number')
#OUTPUT : Its a positive Number
```

if a number is multiple of 3 but not 5 print fizz
if a number is multiple of 5 but not 3 print buzz
if a number is multiple of both print fizz buzz
if a number is not a multiple of both, print not fizz buzz

```
# if a number is multiple of 3  but not 5 print fizz
# if a number is multiple of 5  but not 3 print buzz
# if a number is multiple of both print fizz buzz
# if a number is not a multiple of both, print not fizz buzz
a = int(input('Enter a number: '))
if ((a%3)==0):
  if ((a%5)==0):
    print('fizz buzz')
  else:
    print('fizz')
elif ((a%5)==0):
  print('buzz')
else:
  print('not fizz buzz')

Enter a number: 3
fizz
```

(or)

```
a=int(input('enter a number= '))
if ((a%5==0) or (a%3==0)):
  if ((a%3)==0 and (a%5)!=0):
    print('fuzz')
  elif ((a%5)==0 and (a%3)!=0):
    print('buzz')
  else:
    print('fizz buzz')
else:
  print('a is not divisible by both')

enter a number= 3
fuzz
```

Practice Examples:

Accept only integers, Make a number always positive though given negative

```
Number=int(input('Enter a Number= '))
if (Number<0):
 Number=Number*(-1)
 print(f'Positive value is {Number}')
else:
 print(f'Value is {Number}')
```

Find the largest number among three numbers?

```
a, b, c = 10,100,100
if (a==b==c):
  print("All are equal")
elif (a>=b) and (a>=c):
  if (a==b):
       print("a & b are largest")
  elif (a==c):
   print("a & c are largest")
  else:
     print("a is largest")
elif (b>=a) and (b>=c):
  if (b==c):
    print("b & c are largest")
  else:
    print("b is largest")
```

```
else :
    print("c is largest")
```

Confirm whether given number is a 3 digit number?

```
# Confirm whether the given number is a 3 digit number?
a = 1000
if ((a>=100) and (a<1000)):
  print("True")
else:
  print("False ")
```

Confirm whether given number is of datatype integer and check its a 3 digit number?

```
a = input('enter a number: ')
if (a.isnumeric()):
  a = int(a)
  if (a>99) and (a<1000):
    print(' input has three digit')
  else:
    print(' input does not have three digits')
else:
  print('Invalid input given')
```

Leap Year or not

1. **Completely divisible by 4 but not 100**
2. **completely divisible by 4, 100, 400**

```
a = int(input('Enter a year :'))
if ((a%4)==0):
 if ((a%100)==0):
   if ((a%400)==0):
     print("it is leap year")
   else :
     print("it is not a leap year")
 else:
   print("it is a leap year")
else:
 print("it is not a leap year")
```

Based on the input side length values, find whether its a scalene or Isosceles or Equilateral Triangle?

```
#scalene/isosceles/equilateral triangle?
```

```
A=int(input ('Enter the Value of Side A= '))
B=int(input ('Enter the Value of Side B= '))
C=int(input ('Enter the Value of Side C= '))
if (A==B) or (B==C) or (A==C):
   print("Two Sides are Equal Hence it is isosceles")
elif (A==B==C):
   print("All Sides are Equal hence it is Equilateral")
else:
 print("All Sides are different hence Scalene")
```

Entrance exam permission validation:

- **Students above 75% attendance should be allowed for exam**
- **Students with permission are allowed though less than 75%**

```
a=int(input('No. of total classes : '))
b=int(input('No. of classes you attended : '))

if ((b/a)*100)>=75:
 print('You are allowed to exam')
elif (((b/a)*100)<75):
 c=input('Have you taken permission : \n\nA) Yes    B) No \n')
 if (c=='Yes'):
   print("You are allowed to exam")
 else:
   print('You are not allowed')
```

A shop will give discount of 10% only if you purchase more than 1000 rupees.

Any item costs exactly 100 rupees.

Ask userinput for the number of products purchased and tell the customer the final bill.

```
Quantity=int(input('Enter Quantity= '))
invoice=100*Quantity
if (invoice>=1000):
 invoice=invoice*0.9
 print(f'Final amount after discount is {invoice}')

else:
 print(f'Final amount is INR {invoice}')
 print(f'Please add INR {1000-invoice} to get 10% discount')
```

pass:

Never keep conditions empty, instead use **pass** to avoid getting an error.

Example:

```
A = 2
B = 1
if B > A:
  pass

#OUTPUT : No error and does nothing
```

This will not raise any error though you didn't give any statement to execute after if condition.

Iterative operation:

To Iterate on items individually in a sequence or Repetition of tasks

Loops:

Python has two ways of defining loops. They are-

- for
- while

for loop:

- **for** loop is basically used to have an access to individual items in a list, tuple, strings, set and Dictionaries.
- It is used for iterating over a sequence.
- We can execute a set of statements, once for each item in a list, tuple, set etc.


```
# for loop

# SYNTAX Of for loop
# for (temporarily iterative variable) in (sequence):
  # task/statement
```

To print 1 to 4 numbers in general

```
i = 1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
i = i+1
print(i)
```

for loop using range:

Range is used to print a specific interval of values.

Syntax: **range(start_value, end_value, step_value)**

```
# Example -1
for i in range(10):
    print(i) # prints values from 0 to 9, default start_value = 0
'''
OUTPUT :
0
1
2
3
4
5
6
7
8
9
'''
```

```
# Example -2
for i in range(2,10):
    print(i) # prints values from 2 to 9, default step_value = 1
'''OUTPUT
2
3
4
5
6
7
8
9
'''

# Example - 3
for i in range(2,10,2):
    print(i) # prints values from 2 to 10, step_value = 2
'''
OUTPUT :
2
4
6
8
'''

# Example -4
for i in range(20,10,-2):
    print(i) # prints values from 20 to 10, step_value = -2
'''
OUTPUT :
20
18
16
14
12
'''
```


Note:

If the **start_value < end_value,** values increment, Step_size must be positive.

If the **start_value > end_value,** values decrease, Step_size must be negative.

for loop over Strings:

Example:

```
for x in 'Python':
  print(x)
# Output is each character printed in a new line
'''
OUTPUT :
P
y
t
h
o
n
'''
```

In the above example,

- for - is the Keyword
- x - is a temporary variable for each item, it changes for each iteration with index value.
- in - Membership operator
- Colon : at the end of the for loop line is mandatory
- Indentation is mandatory right after the for loop line.

In python 'for loop' incrementation need not be specified. It is by default plus 1.

Print even numbers between 0 to 20

```
for i in range(20):
  if ((i%2)==0):
    print(i)
```

Usage of range( ) in strings:

```
a = 'Python'
b = len(a)
for x in range(b):
  print(a[x])
```

```
#or

for x in a:
  print(x)

# Output is each character printed in a new line
'''
OUTPUT :
P
y
t
h
o
n
'''
```

for loop over Lists:

Example:

```
alist = ["one", "two", "three"]
for x in alist:
  print(x)

#or

for x in range(len(alist)):
  print(alist[x])

# Output is each item printed in a new line
'''
OUTPUT :
one
two
three
'''
```

In the above list example, 'for loop' iterates over items, but not characters of the items like in strings.

for loop over dictionaries:

Example:

```
mydict = {'hello': 'python','bello':'try','hey':'hello'}
for x in mydict:
   print(x)
   #print(mydict)

# By default for loop prints keys only
# Output is each item printed in a new line

'''
OUTPUT :
hello
bello
hey
'''
```

In the above dictionary example, 'for loop' iterates over keys only, but not values of the items.

Example:

```
mydict = {'hello': 'python', 'bello':'try', 'hey':'hello'}
for x in mydict:
   print(x) # prints key
   print(mydict[x]) # prints value

# By default for loop prints keys only
# Output is each item printed in a new line
'''
OUTPUT :
hello
python
bello
try
hey
hello
'''
```

break:

There might be instances where you need to break the loop without iterating throughout the loop. Such situations, the **break** keyword is used.

Example-1

```
a = 'Hello Python'
b = len(a)
for x in range(b):
  if x == 5:
     break # will stop the loop iterations when it reaches x = 5
  print(a[x])
 # Output is each character printed in a new line
'''
OUTPUT :
H
e
l
l
o

'''
```

Example-2

```
alist = ["one", "two", "three",'four', 'five']
for x in alist:
  if x == 'three':
     break  # will stop the loop iterations when it reaches x =
'three'
  print(x)
# Output is each item printed in a new line
'''
OUTPUT :
one
two
'''
```

Print reverse of the String?

```
# Reverse the String
a = 'hello'

# a[::-1]
for i in a[::-1]:
  print(i,end = '')
```

```
# Alternative Reverse of the String
a='hello'
for i in range(1,len(a)+1):
  print(a[-i],end = "")
```

```
# Alternative Reverse of the String
a = 'Hello'
b = ""
for i in a:
  b = i+b
print(b)
```

```
# Alternative Reverse of the String
a = 'Hello'
d = len(a)
b = ""
for i in a:
  b = b+a[d-1]
  d = d-1
print(b)
```

continue:

You can skip the current iteration using **continue** keyword.

Example-1

```
a = 'Hello Python'
b = len(a)
for x in range(b):
  if x == 6:
    continue  # will skip the current iteration when it reaches x
= 5
  print(a[x])
 # Output is each character printed in a new line
'''
OUTPUT :
H
e
l
l
o
```

```
y
t
h
o
n

'''
```

Example-2

```
alist = ["one", "two", "three",'four', 'five']
for x in alist:
  if x == 'three':
    continue #will skip the current iteration when it reaches
x='three'
  print(x)
# Output is each item printed in a new line

'''
OUTPUT :
one
Two
four
five
'''
```

Nested Loops:

A nested loop is a loop inside a loop.

The whole "inner loop" will be executed once for each iteration of the "outer loop".

It means one iteration of the outer loop is equal to all iterations of the inner loop.

Example:

```
parents =['parent1', 'parent2', 'parent3']
kids = ['baby1', 'baby2', 'baby3']
for x in parents: # outer loop
 print(x)
 for x in kids: # inner loop
   print('Hi',x)

'''
```

```
OUTPUT :
parent1
Hi baby1
Hi baby2
Hi baby3
parent2
Hi baby1
Hi baby2
Hi baby3
parent3
Hi baby1
Hi baby2
Hi baby3
'''
```

```
# Nested Loops
a = ['hi','hello','hey']
b = ['akhil','prateek','indresh']
for i in a:
  for j in b:
    print(i,j)
```

Enumerate function on loops:

```
for i in enumerate(a):
 print(i)

for i,j in enumerate(a):
 print(i,j)


for i,j in enumerate(a,10):
 print(i,j)
```

Practice Problems:

Sum of the first 10 natural numbers?

```
N=int(input("Enter a range of natural num:  "))
sum=0
for i in range (1,11):
   sum=sum+i
```

```
   #print(sum)
print(f"Sum of first {N} numbers is: ", sum)
```

Sum of the first ‘n’ natural numbers?

```
N=int(input("Enter a range of natural num:  "))
sum=0
for i in range (1,(N+1)):
   sum=sum+i
   #print(sum)
print(f"Sum of first {N} numbers is: ", sum)
```

Print Table 10?

```
table=0
for i in range (1,11):
  table=i*10
  print(f"10 x {i} = {table}")
```

Print Table 2 to Table 10?

```
start=int(input("Enter a Num for starting range : "))
End=int(input("Enter a Num for end of range : "))

for i in range(start,(End+1)):
    print(f"Print Table of {i}")
    for j in range(1,11):
        table=i*j
        print(i,'X',j,'=', table)
```

Check whether datatype is int or not without using any built-in function
Print True if numeric else False if found Non-numeric

```
a=input('enter the value: ')
for i in a:
 if (int(i) in range(0,10)):
   print('True-numeric')
 else:
   print('false-"non-numeric"')
```

(or)

```
a = '0123456789'
b = input('enter the value: ') #124a
c = 0
for i in b:
  if i in a:
    c = c+1

if (c == len(b)):
  print('Is Numeric')
else:
  print('Is non-numeric')
```

pass:

Never keep loops empty, instead use **pass** to avoid getting an error.

Example:

```
for x in (0, 1, 2):
  pass
# OUTPUT : No error
```

The above code will not raise any error even if you haven't done any operation in ‘for loop’ because of the ‘pass’ keyword.

while loop:

As long as your condition is True, statements are executed in the while loop.

```
# SYNTAX
# initilisation (start value)
# while (condition for termination):
   #statement
```

```
for i in range(1,11):
  print(i)
```

```
i=1
while i in range(1,11):
```

```
  print(i)
  i+=1
```

'while loop' on strings-

Example:

```
str1 = 'Iamhappy'
print('length is:',len(str1))
# while loop
i=0
while i < len(str1):
 print(str1[i])
 i = i+1

'''
OUTPUT :
length is: 8
I
a
m
h
a
p
p
y
'''
```

'while loop' on list-

Example:

```
sed = ['a','b','c','d']
# print('length:',len(sed))
i=0
while i<(len(sed)):
   print(sed[i])
   i=i+1
'''
OUTPUT :
a
b
```

```
c
d
'''
```

'while loop' on Range-

Example:

```
x=2
while x in range(2,10):
 x= x+2
 print(x)
'''
OUTPUT :
4
6
8
10
'''
```

break :

Example:

```
str2 = 'pineapple'
i = 0
while i<len(str2):
 if str2[i] == 'e':
   # i = i+1
   break
 print(str2[i])
 i = i+1

'''
OUTPUT :
p
i
n
'''
```

continue:

Example:

```
# Take a string, skip if it is 'e' or else print.
str2 = 'pineapple'
i = 0
while i<len(str2):
 if str2[i] == 'e':
   i = i+1
   continue
 print(str2[i])
 i = i+1

'''
OUTPUT :
p
i
n
a
p
p
l
'''
```

Print 1 to 10 numbers except number 5?

```
#Print all except 5
i=1
while i in range(1,11):
   if (i==5):
     i=i+1
     continue
   else :
     print(i)
     i=i+1
```

while-else:

With the **else** statement we can run a block of code once when the condition no longer is True:

Example:

```
x=2
while x in range(2,10):
```

```
  x= x+2
  print(x)
else :
 print('I am done')
'''
OUTPUT :
4
6
8
10
I am done
'''
```

Practice Problems:

Print any table using while loop?

```
i=1
num=int(input("Enter Number to print the table : "))
while i in range (1,11):
 print(f"{i}X{num}={i*num}")
 i=i+1
```

Print multiple tables using while loop?

```
#Print Table between 2 Numbers
i=1
num1=int(input("Enter Starting Series to print the table : "))
num2=int(input("Enter Ending Series to print the table : "))
while i in range(1,11) :
 print(f"{num1}X{i}={i*num1} ")
 i=i+1
 if (i>10) and (num1<num2):# Completes the last series
   num1=num1+1             # Reassign Num1 to next number
   print(f"Series of {num1}")
   i=1                     # initialize i to 1 to print the next
series
```

Print count of a character using while loop without using count builtin method?

```
a = 'pythonpy'
```

```
# a.count('p')

b = len(a)
count = 0
while count<b:
  if a[count]=='p':
    print(a[count])
  count+=1
```

(or)

```
a = input('Please enter a string: ')
b = input('Please enter the letter to know the count: ')
count = 0
index = 0
l = len(a)
while index<l:
 if (a[index]==b):
   count+=1
 index+=1
print(count)
```

Write a python program to check whether given number is a prime number or composite number?

```
num = int(input('Enter a number to check prime/not: '))
a = 2
count = 0
while a<num:
 if ((num % a)==0):
   count+=1
 a+=1
 if (count>0):
 print('It is a Composite number')
elif ((num==0) or (num==1)):
 print('It is neither prime nor composite')
else:
 print('It is a prime number')
```

Prime or composite number using for loop?

```
# Program to check if a number is prime or not

# To take input from the user
num = int(input("Enter a number: "))

# define a flag variable
flag = False

# prime numbers are greater than 1
if num > 1:
    # check for factors
    for i in range(2, num):
        if (num % i) == 0:
            # if factor is found, set flag to True
            flag = True
            # break out of loop
            break

# check if flag is True
if flag:
    print(num, "is not a prime number")
else:
    print(num, "is a prime number")
```

Patterns

```
'''Bring this output using loops.
*
**
***
****
*****'''
for i in range(1,6):
  print('*'*i)

#Output
*
**
***
****
*****
```

```
i=1
while i in range(1,7):
  print('*'*i)
  i=i+1

#Output
*
**
***
****
*****
******
```

```
i=1
MaxStarts=15
while i in range(1,MaxStarts):
  print('*'*(MaxStarts-i))
  i=i+1

#Output
**************
*************
************
***********
**********
*********
********
*******
******
*****
****
***
**
*
```

```
MaxStarts=7
for i in range(1,7):
  print('*'*(MaxStarts-i))

******
*****
****
***
**
*
```

```
i = 4
for j in range(1,10,2):
  print(' '*i,'*'*j)
  i=i-1

#Output
    *
    ***
   *****
  *******
 *********
```

```
#MAKE A PYRAMID OF GIVEN NUMBERS OF ROWS.
c=1
b=int(input("Enter Numbers of Rows : "))
j=1
for i in range(1,b+1):
 print(((b-i)*" ")+("*"*c)+((b-i)*" "))
 c=c+2

#Output
Enter Numbers of Rows : 5
    *
   ***
  *****
 *******
*********
```

```
# ********* 9
#  *******  7
#   *****   5
#    ***    3
#     *     1
#MAKE A INVERTED PYRAMID
b=int(input("Number of rows: "))
c=((2*(b+1))-1)-2
for i in range (0,b+1):
  print((i*' ')+(c*'*')+(i*' '))
  c=c-2

#Output
Number of rows: 5
*********
 *******
  *****
```

```
   ***
    *
```

```
#     *
#    ***
#   *****
#  *******
# *********
#  *******
#   *****
#    ***
#     *
def twinpyramid():
 MaxStars=int(input("Enter Max Starts in odd nums: "))
 Maxrows=MaxStars//2
 print(f'Rows on both side {Maxrows}')
 Count=1
 j=0 # Print Spaces counts
 for i in range(1,Maxrows+2):
  print(((Maxrows+1-i)*" ")+("*"*Count)+((Maxrows+1-i)*" "))
  Count=Count+2
 for k in range(1,MaxStars+2):
  MaxStars=MaxStars-2
  j=j+1
  print(((j)*" ")+("*"*MaxStars)+((j)*" "))
twinpyramid()

#Output
Enter Max Starts in odd nums: 5
Rows on both side 2
  *
 ***
*****
 ***
  *
```



```
R=int(input("Enter numbers of rows: "))
i=1
while i in range(1,R+1):
  print('1'*i)
  i=i+1

#Output
```

```
Enter numbers of rows: 5
1
11
111
1111
11111
```

```
R=int(input("Enter numbers of rows print 1: "))
i=1
C=R
while i in range(1,R+1):
  print('1'*C)
  C=C-1
  i=i+1

#Output
Enter numbers of rows print 1: 5
11111
1111
111
11
1
```

```
R=int(input("Enter numbers of rows: "))
i=1
j=0
while i in range(1,R+1):
  Value=j+1
  V=str(Value)
  print(V*i)
  i=i+1
  j=j+1

#Output
Enter numbers of rows: 5
1
22
333
4444
55555
```

```
R=int(input("Enter numbers of rows: "))
i=1
j=0
while i in range(1,R+1):
  Value=R+1-i
  V=str(Value)
  print(V*Value)
  i=i+1

#Output
Enter numbers of rows: 5
55555
4444
333
22
1
```

```
i=1
v=''
R=int(input("Enter number of rows: "))
while i in range(1,R+1):
  v=v+str(i)
  print(v)
  i=i+1

#Output
Enter number of rows: 5
1
12
123
1234
12345
```

```
b=7
for i in range(1,6):
  b=b-1
  v=''
  for j in range(1,b):
    v=v+str(j)
  print(v)

#Output
12345
1234
123
```

```
12
1
```

```
b=7
for i in range(1,6):
  b=b-1
  v=''
  for j in range(1,b):
    v=v+str(j)+' '
  print(v)

#Output
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

```
b=7
for i in range(1,6):
  b=b-1
  v=''
  for j in range(1,b):
    v=v+str(j)
    z=v[::-1]
  print(z)

#Output
54321
4321
321
21
1
```

```
a = ['A','B','C','D','E']
out = ''
for i in range(0,5):
  out+=a[i]
  print(out)

#Output
A
```

```
AB
ABC
ABCD
ABCDE
```

```
n = int(input('Enter the number of rows: '))
for i in range(1, n+1):
  print(" "*(n-1), (chr(64+1)+' ')*i)

Enter the number of rows: 5
      A
      A A
      A A A
      A A A A
      A A A A A
```

```
n = 5


for i in range(1,n+1):
    print( i * "0")


for i in range(1,n+1):
    print((n-i) * " " + i * "0" )


for i in range(1,n+1):
    print((n-i) * " " + i * " *" )
```


Functions:

Function is a container having a set of executable statements which can be used whenever called upon.

Types of Functions

1. User-defined functions
2. Lambda functions(Python specific)
3. Recursions functions
4. Pre-defined functions(Built-in)

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

A function can return data as a result.

User-defined functions:

A function is defined using the **def** keyword.

Example:

```
def hello():
  print('Hello world')
hello()
#OUTPUT: Hello world
```

To call a function, use the function name followed by parenthesis:

Parameter / Argument:

Parameter : Value passed while defining a function.

Argument : Value passed while calling back a function.

- Parameters and arguments are ordered by default
- No. of Parameters must be same as No. of Arguments
- *args
- *kwargs

What is the difference between Parameters and Arguments?

Function **parameters** are the names listed in the function's definition. Function **arguments** are the real values passed to the function. **Parameters** are initialized to the values of the **arguments** supplied.

```
def Greetings():
 print('HI Everyone')

Greetings()
```

Example:

```
def call_name(name):
  print('I am ',name,' i will learn python')
```

```
call_name('Rohith')
#OUTPUT: I am  Rohith  i will learn python
```

In the above example, **name** is the parameter and **Rohith** is the argument.

```
def Naming(name):
  print('Hi ',name)
  print('Hello ',name)
  print('Hola ',name)

Naming(Akhil)

#Output
Hi  Akhil
Hello  Akhil
Hola  Akhil
```

Note: Function must call the correct number of arguments as it expects. Having less or more will raise an error.

Example:

```
def Afunction(firstname, lastname):
  print(firstname + " " + lastname)
Afunction("Akhil", "Teja")

#OUTPUT: Akhil Teja
```

Default Parameter Value:

If we call the function without argument, it uses the default value:

Example:

```
def Afunction(name="Akhil"):
  print('My Name is ', name)
Afunction("john")
Afunction( ) # default parameter helps here
Afunction("Teja")

'''OUTPUT:
My Name is  john
```

```
My Name is  Akhil
My Name is  Teja
'''
```

Arbitrary Arguments(*args):

If you are not sure about how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly

Example1:

```
def Afunction(*name):
  print('My Name is ', name[2])
Afunction("Akhil", "John", "Teja")

#OUTPUT:  My Name is  Teja
```

Example2:

```
def exam2(*name,name1='ind',name2='joy'):
  print(f'hi {name[4]}')
  print(f'hi {name[5]}')
  print(f'hi {name1}')
  print(f'hi {name2}')

exam2(john,'akhil',sam,'jab','lion','tiger')
```

Keyword Arguments (*kwargs):

You can also send arguments with the key = value syntax while calling back a function,

Note: This way the order of the arguments does not matter.

Example:

```
def Afunction(name2, name1, name3):
  print('My Name is', name2)
Afunction(name1 = "Akhil", name2 = "sai", name3 = "Teja")

#OUTPUT: My Name is sai
```

Example:

```
def exam2(name1,name4, name2='jey',name3='keee'):
  print(f'hi {name1}')
  print(f'hi {name2}')
  print(f'hi {name3}')

exam2('joy','jab',name2='prateek',name3='jay')
```

Note:
The order of passing parameters must be like below:
Regular parameter, *args, default parameters

Example:

```
def exam2(name1,*name2,name3='jiii',name4='teja'):
  print(f'hi {name1}')
  print(f'hi {name2}')
  print(f'hi {name3}')
  print(f'hi {name4}')

exam2('pratik','akhil','tailor')

#output
hi pratik
hi ('akhil', 'tailor')
hi jiii
hi teja
```

In the above example,

1. Regular parameter is **name1**
2. *args is ***name2**
3. Defaults are name3, name4

return keyword:
Till now we had been printing the statements for the output. You can also just **return** the output in functions and can be printed during the function call back.

Example:

```
a=2
def abc(a):
  return a+1 #returns expression value, functions execution halts
abc(a) #prints value
```

Example:

```
a=2
def abc(a):
  return a+1 #
  print('hi') #hi is not printed as function returned in the above line

abc(a)
```

Example:

```
a=2
#prints both print functions
def abc(a):
  print(a+1)
  print('hi')
abc(a)
```

Example:

```
a=2
def abc(a):
  return a+1 #returns expression value, functions execution halts
  return a+10  # error,You cannot return more than once in a function

abc(a)
```

Example:

```
def Afunction(name="Akhil"):
  return 'My Name is ', name
print(Afunction("john"))
print(Afunction( )) # default parameter helps here
print(Afunction("Teja"))
```

And as we already know about **pass** , it is used to skip the function without raising an error if left empty.

Write a function to calculate cube of a number?

```
def ccube(a):
```

```
 print(a**3)

ccube(int(input('Enter a number :- ')))
```

Example:

```
def Afunction():
  pass
# does nothing, no error
```

---

12. Lambda function:

- A lambda function is a shorter format of a function also called an "**anonymous function**".
- It accepts any number of arguments but will have **only one expression.**

Syntax: **lambda (arguments) : (expression)**

Here **lambda** is a keyword.

General way of writing a function below:

```
a = int(input('a number please: '))
def ranc(b):
  print(b+10)
ranc(a)
```

Rewriting the above function using Lambda function

```
# lambda parameter: expression
# b = int(input('a number: '))
d = lambda b: b+10
d(6)
```

Example:

```
# General function:
def Afunction(val):
   return val * 10
```

```
print(Afunction(10))
# OUTPUT : 100

# lambda function
x = lambda val : val *10
print(x(10))
# OUTPUT : 100
```

Lambda function will have only one expression:

```
# General function:
def Afunction(base, power):
   return base ** power
print(Afunction(10,2))
# OUTPUT : 100

# lambda function
x = lambda base, power : base ** power
print(x(10,2))
# OUTPUT : 100
```

Example:

```
add = lambda a,b : a+b
sub = lambda a,b : a-b
mul = lambda a,b : a*b
div = lambda a,b : a/b

print(add(10,20))
print(sub(10,20))
print(mul(10,20))
print(div(10,20))
```


Lambda function inside a function is always helpful:

Example:

```
def func(n):
  return lambda a : a * n

doubled = func(2)
print(doubled(11))
# OUTPUT : 22
```

Note: Use lambda functions when an anonymous function is required for a short period of time.

Find whether a number is Even or Odd using Lambda function?

```
# lambda func. using if-else
b=int(input('enter a number: '))

i=lambda b: print('given no. even.') if ((b%2)==0) else print('it
is odd')    ## else is mandatory

i(b)
```

Find whether a number is a Positive or Negative or Zero using Lambda function? (if - elif - else)

```
# lambda func. using if-elseif

a=int(input('enter a number: '))

b=lambda a:print('it is positive') if (a>0) else print('it is
negative') if (a<0) else print('it is zero')

b(a)
```

Find whether a number is a Positive or Negative or Zero using Lambda function? (Nested if - else)

```
# lambda func. using nested if-else
a=0

b=lambda a: print('it is negative') if (a<0) else (print('it is
positive') if (a>0) else print('it is zero'))

b(a)
```

Find whether a number is a Multiple of 3 or 4 using Lambda function?

```
# Write a lambda function To check whether a number is a multiple
of 4 or 3
a = lambda b: print('multiple of 4') if (b%4==0) else
print('multiple of 3') if (b%3==0) else print('not multiple of 4
```

```
and 3')
a(15)
```

Recursions

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as **recursive functions**.

- Recursion is the process of defining something in terms of itself like a finite loop.
- A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Example:

Printing 0 to 10 numbers using for loop

```
for i in range(11):
  print(i)
```

Printing 0 to 10 numbers using while loop

```
i=0
while i<11:
  print(i)
  i+=1
```

Printing 0 to 10 numbers using recursive function

```
def rec(x):
  if x==11:
    return
  else:
    print(x)
    return rec(x+1)
rec(0)
```

Printing 10 to 0 numbers using recursive function

```
def rec2(x):
  if x<0:
    return
  else:
```

```
    print(x)
    return rec2(x-1)
rec2(10)
```

Print Reverse of the string using recursions?

```
def reverse(a):
 if len(a)==0:
   return a
 else:
   return reverse(a[1:])+a[0]

reverse('python')
```

Find the sum of the digits of the given number using recursion?

For example:
Sum of given number:
123 = 1+2+3 =6
346 = 3+4+6 = 13

```
n = 7234
def sum(x):
 if x==0:
   return 0
 else:
   return ((x%10)+sum(int(x/10)))
sum(n)
```

If the input is a string datatype-

```
def total(a):
  if len(a)==0:
        return 0
  else:
        return int(a[0])+total(a[1:])
total(input('Enter a Value :- '))
```

Print factorial of a number using recursions?

For example:

Factorial of the given number:
0! = 1
1! = 1
2! = 2x1 = 2
3! = 3x2x1 = 6
4! = 4x3x2x1 = 24
5! = 5x4x3x2x1 = 120
and so on.

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

What is happening above?

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

Print Fibinocci series using recursions?

For example:
Factorial of the given number:
0! = 1
1! = 1
2! = 2x1 = 2
3! = 3x2x1 = 6
4! = 4x3x2x1 = 24
5! = 5x4x3x2x1 = 120
and so on.

```
def fibi(x):
```

```
  if x<=1 :
    return 1
  else:
    return fibi(x-2)+fibi(x-1)
fibi(5)

#To print the fibonacci series
N = 10
for i in range(N):
  print(fibi(i))
```

Armstrong or Narcissistic Number

- A number is said to be Armstrong or Narcissistic Number if it is equal to the sum of its own digits raised to the power of the number of digits in a given number.
- All Single digit numbers(0 to 9) are Armstrong numbers.

153 = $1^3+5^3+3^3$ = 153
370 = $3^3+7^3+0^3$ = 370
407 = $4^3+0^3+7^3$ = 407
1634 = $1^4+6^4+3^4+4^4$ = 1634
8208 = $8^4+2^4+0^4+8^4$ = 8208
And so on.

```
b=input()
def armstrong(a):
 if (len(a)==0):
   return 0
 else:
   return ((int(a[0]))**len(a)+armstrong(a[1:]))

# armstrong(b)
if (armstrong(b)==int(b)):
 print('given no is amstrong')
else:
 print('given no is amstrong')
```

Advantages of Recursion:

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Scoping:

Local Scope:
A variable declared inside a function is part of the function's local scope but can only be utilised inside that function.

Example:

```
def dummy():
    a = 2
    print(a) #Prints output of a value

dummy()
print(a) #Gives an error because a is a local scope variable
```

The variable **a** is not accessible outside of the function, however it is accessible within the function:

Example:

```
def dummy():
    a = 2
    a += 2 #can do any type of assignments or operations on a
    print(a) #Prints 4 as an output of a value

dummy()
print(a) #Gives an error because a is a local scope variable
```

Global Scope:
A global variable in Python is a variable defined outside of the function or in the global scope.
A global variable can be accessed both inside and outside of the function.

Example:

```
a = 2
def dummy():
   print(a) #Prints output of a value

dummy()
print(a) #Prints output of a value
```

In the above code variable **a** gets printed inside and outside of the function as its a global variable.

You cannot change the global variable directly inside the local scope like below- It raises an UnboundedLocalError.

Example:

```
a = 2
def dummy():
   a += 2 # error, cannot be referenced before assignment
   print(a)

dummy()
print(a)
```

In case you want to change the local variable, you have to make it a global variable using "global" keyword.

Example:

```
a = 2
def dummy():
   global a #declaring variable a as global variable
   a = 3
   print(a) #prints modified global variable a value i.e., 3

dummy()
print(a) #prints modified global variable a value i.e., 3
```

Example:

```
a = 2
def dummy():
   global a #declaring variable a as global variable
   a += 2 # will modify the a value
   print(a) #prints modified a value i.e., 4
```

```
dummy()
print(a) #prints modified a value i.e., 4
```

The changes that we do in local scope also modifies outside the function because variable a is converted as global variable.

Global & Local :

Example:

```
a = 2 # Global variable
def dummy():
   a = 3 # Local variable
   print(a) #prints local varaible a value i.e., 3
   a += 2 # will modify the a value in local scope
   print(a) #prints modified a value i.e., 5

dummy()

print(a) #prints global varaible a value i.e., 2
a += 2 # will modify the a value in global scope
print(a) #prints modified a value i.e., 4
```

Example:

```
a=2 #global variable a
def b():
  a=3 #local variable a in the scope of b function
  def c():
    a=4 #local variable a in the scope of c function
    print(a) #prints local variable a value i.e., 4
  c()
  print(a) #prints local variable a value i.e., 3

print(a) #prints global variable a value
b()
```

Non-local variable:

Nonlocal variables are utilised in nested functions with an undefined local scope. This indicates the variable can't be in both the local and global scopes.

Example:

```
def dummy1():
    a = 2

    def dummy2():
        global a
        a = 3
        print("dummy2:", a) #prints 3

    dummy2()
    print("dummy1:", a)#prints 2, global doesn't affect this

dummy1()
```

Example:

```
def dummy1():
    a = 2

    def dummy2():
        nonlocal a
        a = 3
        print("dummy2:", a) #prints 3

    dummy2()
    print("dummy1:", a)#prints 3

dummy1()
```


Few Built-in Functions:

abs( ):
Mostly used to convert a negative value to positive value, its because of the following formula being applied.

→ sqrt((real)**2 + (imag)**2)

Example:

```
a = -3
#((-3)**2)**(1/2)
```

```
abs(a)

a = -5.67
#((-5.67)**2)**(1/2)
abs(a)

a = 3-4j
#(3**2 + (-4)**2)**(1/2)
abs(a)
```

any( ):

Returns True if atleast one of the iterable items is true.

```
a = [0,False,'0']
any(a)
```

all( ):

Returns True if all of the iterable items are true.

```
a = [0,False,'0']
all(a)
```

Number System functions:

We know that, By default Interpreter accepts decimal values.

Binary:

- It is represented with 0b as prefix
- **bin()** Converts given input number system to binary number system value.

Example:

```
0b10101
```

Octal :

- It is represented by 0o as prefix
- **oct()** Converts given input number system to binary number system value.

Example:

```
0o123
```

Hexadecimal:

- It is represented by0x as prefix

- **hex()** Converts given input number system to binary number system value.

Example:

```
0x12AB
```

Conversions:

Decimal Conversions:

int() function is used to convert any number system to decimal number system.

```
# binary to decimal
int('1000',2)
#Octal to decimal
int('12',8)
#hexadecimal to decimal
int('A',16)
```

Binary Conversions:

bin() function is used to convert any number system to bianry number system.

Example:

```
# decimal to binary
bin(8)

# octal to binary
bin(0o123)

# hexadecimal to binary
bin(0xAB)
```

Octal Conversions:

oct() function is used to convert any number system to octal number system.

Example:

```
# decimal to octal
oct(8)

# binary to octal
oct(0b1111)
```

```
# hexadecimal to octal
oct(0xAB)
```

Hexadecimal Conversions:

hex() function is used to convert any number system to octal number system.

Example:

```
# Decimal to hexadecimal
hex(8)

# binary to hexadecimal
hex(0b101)

# octal to hexadecimal
hex(0o123)
```

ASCII (American Standard Code for Information Interchange):

For every character that's in use has a dedicated decimal number assigned for the machine to understand in binary format.

Reference:

https://computersciencewiki.org/images/3/3d/Ascii_table.png

Character to ASCII conversion:

ord() function converts the given input number to ASCII number.

```
ord('A')
ord('z')
ord('&')
```

ASCII to Character conversion:

chr() function converts the given input ASCII number to Character.

```
chr(60)
chr(100)
chr(77)
```

iter():

Used as an Iterator

next()

used to iterate one item at a time

```
a = ['a','b','c','d']
b = iter(a)
print(next(b))
print(next(b))
print(next(b))
print(next(b))
```

map():
The map() method produces an iterator after applying a provided function to each item of an iterable (list, tuple, etc.).

Syntax: map(function,input iterable)

```
def square(n):
  return n * n

a = [1,2,3,4,5,6]
list(map(square,a))

#using lambda function
b = (map(lambda i:i**2,a))
list(b)
```

filter():
The filter() method selects entries from a list, tuple, or any iterable for which a function returns True.
Syntax: filter(function,input iterable)

```
def even_odd(n):
 if n % 2 == 0:
     return True

a = [1,2,3,4,5,6]
list(filter(even_odd,a))

#using lambda function
b = filter(lambda i:(i%2)==0, a)
list(b)
```

13. List Comprehension:

List Comprehension is a handful syntax when you want a new list by using ‘for loop’ and ‘if-else conditions’ together usually.

- To begin with, you're condensing three lines of code into one.
- List comprehension is quicker since Python allocates the list's memory first before adding the entries, avoiding the need to enlarge the list at runtime.
- It will also save you time on 'append' calls, which are little but pile up.
- Finally, code written with comprehensions is seen to be more 'Pythonic,' according to Python's style rules.

An example with just a ‘for loop’ below:

Syntax: **[expression for loop]**

Example:

```
# General list appending
letters = []
for letter in 'I know python':
    letters.append(letter)
print(letters)
# OUTPUT : ['I',' ','k','n','o','w',' ','p','y','t','h','o','n']

# List Comprehension
letters = [ letter for letter in 'I know python' ]
print( letters)
# OUTPUT : ['I',' ','k','n','o','w',' ','p','y','t','h','o','n']
```

Example:

Print numbers between 0 to 10 using list comprehension?

```
# [(expression) (for loop)]
[i for i in range(0,11)]

#OUTPUT
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Example:

Print Cube of the numbers between 0 to 10 using list comprehension?

```
[i**(3) for i in range(0,11)]
```

```
#OUTPUT
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Example:
Print characters of a given string as an individual item of the list?

```
a = 'python'
print(list(a))

#is same as below code

a = [i for i in a]
print(a)
```

Generally, if you want to check the presence of a value in a list, we use a for loop to iterate over items of an array and if condition to check the value like below:

Syntax : **[expression for item in iterable if condition == True]**

Example:

```
# If condition
# [(expression) (for loop) (if condition)]
[i for i in range(51) if ((i%5)==0)]

[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

To find even numbers in a range of values?

Example:

```
# General procedure
even_numbers = []
for x in range(20):
    if x %2 == 0:
        even_numbers.append(x)
print(even_numbers)
# OUTPUT : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# Using List comprehension
even_numbers = [ x for x in range(20) if x % 2 == 0]
```

```
print(even_numbers)
# OUTPUT : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

if - else:

Syntax: **[if expression if condition else else expression for loop]**

Example:

```
# If else condition
# Please swap the positions of for loop and if-else condition
here:
# [(expression) (if condition) (else condition) (for loop) ]
[i if ((i%5)==0) else 0 for i in range(10)]

#Output
[0, 0, 0, 0, 0, 5, 0, 0, 0, 0]
```

Print all even numbers as 'bow' and all odd numbers as 'meow' less than 20?

```
['bow' if (i%2==0) else 'meow'  for i in range(1,20) ]
```

if - elif - else List Comprehension:

Syntax: **[if expression if condition else elif expression if condition else else expression for loop]**

Example:
Print whether the given range of numbers are negative or zero or positive in a list?

```
b = []
# Regular if-elif-else
for a in range(-5,5):
 if a>0:
   b.append('positive')
 elif a<0:
   b.append('negative')
 else:
    b.append('zero')

print(b)
```

List Comprehension way of doing the above code-

```
['positive' if (a>0) else 'Negative' if (a<0) else 'zero' for a in range(-5,5)]
```

Nested if condition:

Syntax: **[expression for loop if condition if condition]**

Example:
Print values less than 10 which are multiples of both 2 and 3 ?

```
# 1,100
a = []
for i in range(10):
  if ((i%2)==0):
    if ((i%3)==0):
      a.append(i)

print(a)

#Output
[0, 6]
```

List Comprehension way of doing the above code-

```
# Nested If
# [(expression) (for loop) (if condition) (if condition)]
[ i for i in range(10) if ((i%2)==0) if ((i%3)==0) ]

#Output
[0, 6]
```

Example:
Print fizzbuzz for the multiples of both 5 and 3 in list comprehension ?

```
#Regular way
a = []
for i in range(1,50):
 if ((i%5)==0):
   if ((i%3)==0):
     a.append('fizzbuzz '+f'{i}')
print(a)

#List Comprehension way of doing it
[f'fizzbuzz {i}' for i in range(1,50)if ((i%5)==0) if ((i%3)==0) ]
```

Nested for loop list comprehension:

Syntax: **[expression for loop for loop]**

```
#Regular way
a = [1,2,3,4]
for i in a:
 for j in a:
   print(i,j)

#List comprehension way of doing it
[(i,j) for i in a for j in a]
```

Nested List Comprehension:

Example:
Build a Matrix using List comprehension?

```
# Nested List Comprehension
a = [[1,2],[3,4]]
[[i[r] for i in a] for r in range(2)]

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

Key Points:

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating lists.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.

File Operations:

Storing the data in memory addresses for futuristic purposes creates a **file.**

How to do operations on file using Python?

We must first open a file before we can read from or write to it. When we're finished, it has to be closed so that the file's resources may be released.

As a result, a file operation in Python is performed in the following order:

1. Open the file

2. You can either read or write or append(perform operation)
3. Close the file

Opening of a file using python function open:

open() takes filename as input and the format mode to open with.

```
f = open("test.txt") #To open a file
# By default file opens with "read" mode
```

There are few important modes to open a file.

- r - opens in read mode
- w - opens in write mode, in case file doesn't exist, it creates one
- x - opens in write mode, in case file doesn't exist, it fails to do
- a - opens in append mode, appends to the already existing content, in case file doesn't exist, it creates one
- t - opens in text mode which is by default
- b - opens in binary mode

r - reading the file

```
f = open('/content/test1.txt',’r’) #opening a file in read mode by default
print(f.read()) #read the file
f.close() #closes the file preventing further edits

#Output file
I am a test file content in test1.txt
```

r - reading the limited file

Pointer moves while reading thef file.

```
f = open('/content/test1.txt') #opening a file in read mode by default
print(f.read(12)) #read only first 12 characters of the file
print(f.read(12)) #read next 12 characters of the file
f.close() #closes the file preventing further edits

#Output
I am a test
```

```
file content
```

Shorter syntax for file operation which closes the file implicitly-

```
with open('/content/test1.txt') as f: #opening a file in read mode by default
  print(f.read(12))
#The above sentence closes the file by default after the operation

#Output
I am a test
```

w - writing the file

- If you open a file which is not there yet, a Newfile will be created.
- If you open a file which is already existing, It overwrites the content.

```
f = open('/content/test2.txt','w') #opening a file in write mode
f.write('I am test2 file') #writes content in test2.txt
f.close() #closes the file preventing further edits
```

Shorter syntax for file operation which closes the file implicitly-

```
with open('/content/test2.txt','w') as f:
  f.write('I am test2 file') #writes the content in test2 file
#The above sentence closes the file by default after the operation
```

x - writing the file(Exclusive creation)

- If you try write a file which is already there, operation fails.
- If a file is not existing, It writes the content.

```
f = open('/content/test4.txt','x') #opening a file in write mode
f.write('I am test4 file') #writes content in test2.txt
f.close() #closes the file preventing further edits
```

Shorter syntax for file operation which closes the file implicitly-

```
with open('/content/test4.txt','x') as f:
  f.write('I am test4 file') #writes the content in test4 file
#The above sentence closes the file by default after the operation
```

a - writing the file

- If you open a file which is not there yet, a Newfile will be created.
- If you open a file which is already existing, It concatenates to the existing content.

write() function itself is used for appending too with just a change of mode.

Let's assume that we already have a file called test2.txt which has following content.

"I am content from test2 file"

Using append mode will concatenates to the existing file.

```
f = open('/content/test2.txt','a') #opening a file in append mode
f.write('\n I am also a test2 file')
f.close() #closes the file preventing further edits

#output
I am content from test2 file
I am also a test2 file
```

Shorter syntax for file operation which closes the file implicitly-

```
with open('/content/test2.txt','a') as f:
  f.write('\nI am a test4 file text')
#The above sentence closes the file by default after the operation
```

readline, readlines
readline - return one line at a time of the file
readlines - returns all the lines at a time of the file in a list format

Let's assume that we already have a file called test2.txt which has following content.

"I am content1 from test2 file

I am content2 from test2 file

I am content3 from test2 file"

readline():
Example:

```
# Shorter syntax for file operation without closing syntax
```

```
with open('/content/test2.txt','r') as f: #opening a file in read
mode by default
  print(f.readline())

#output
'I am content1 from test2 file\n'
```

readlines():
Example:

```
# Shorter syntax for file operation without closing syntax
with open('/content/test2.txt','r') as f: #opening a file in read
mode by default
  print(f.readlines())

#output
['I am content1 from test2 file\n', 'I am content2 from test2
file\n', 'I am content3 from test2 file\n']
```

A for loop may be used to read a file line by line. This is both effective and quick.

Let's assume that we already have a file called test2.txt which is stored in '**f**' has following content.

"I am content1 from test2 file

I am content2 from test2 file

I am content3 from test2 file"

```
f = open('/content/test2.txt','r')
for i in f:
  print(i)

#output
I am content1 from test2 file
I am content2 from test2 file
I am content3 from test2 file
```

writelines():

Writes multiple lines at a time to the file if given in list whereas each item is one line at a time.

```
# Shorter syntax for file operation without closing syntax
with open('/content/test6.txt','w') as f: #opening a file in read mode by default
  f.writelines(['i am line1','\ni am line2','\ni am line3'])
```

Exception Handling:

When writing the code, we might make mistakes that cause errors when we try to run it. When a Python programme meets an unhandled error, it terminates. These errors can be divided into two categories:

1. Syntax error
2. Logical error(exceptions)

Example:

```
# Syntax error
if  :
   ^
  print('hey')

#Output
SyntaxError: invalid syntax
```

Logical Errors:
Exceptions or logical errors are setbacks that occur at runtime once the program is syntactically verified.
Example:

```
# logical error
print(dummy)

#Output
ValueError: name 'dummy' is not defined
```

List of Errors:

```
print(dir(locals()['__builtins__']))

#List of errors
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', 'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError',
```

```
'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError',
'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError',
'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError', 'ReferenceError',
'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopAsyncIteration', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__',
'__build_class__', '__debug__', '__doc__', '__import__',
'__loader__', '__name__', '__package__', '__spec__', 'abs', 'all',
'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes',
'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'display',
'divmod', 'dreload', 'enumerate', 'eval', 'exec', 'execfile',
'filter', 'float', 'format', 'frozenset', 'get_ipython',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min',
'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'range', 'repr', 'reversed', 'round', 'runfile',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```


Exception Handling:

Exception handling is done with below given control flow.

try→ except → else → finally

try→ else → finally (Without any error)

try→ except (With any error)
Exceptions in Python could be well addressed with the **try** statement.

The **try** clause contains the necessary operation that might cause an exception.
The **except** clause contains the code that handles exceptions.
The **else** block lets you execute code when there is no error.
The **finally** block lets you execute code, regardless of the result of the try- and except blocks.

As a result, after we've caught the exception, we may pick which actions to conduct.

else and **finally** are totally optional blocks whether to extend the logic or not.

In an ideal scenario lets assume there is no error at all, Here's an easy example.

```
# WITHOUT ANY ERROR
try:
  print('i am try block')
except:
  print('i am except block')
else:
  print('i am else block')
finally:
  print('i am finally block')

#Output
i am try block
i am else block
i am finally block
```

In the above code, as there was no error occurred, it didnt enter into except block.
When there is an error, exception catches it and tries to resolve.

Example:

```
# WITH ERROR
try:
  print('i am try block',dummy)#dummy is an undefined variable
except:
  print('i am except block')
else:
  print('i am else block')
finally:
  print('i am finally block')
```

```
# Output
i am try block
i am except block
```

We can also customize the error handling blocks like below-

Example:

```
try:
  print('i am try block')
  x = 12/0  #error with zero division
except SyntaxError:
  print(' i am a syntax error')
except TypeError:
  print(' i am a TypeError error')
except ZeroDivisionError:
  print(' i am a ZeroDivisionError')
except ValueError:
  print(' i am a Value error')

# Output
i am try block
i am a ZeroDivisionError
```

We can also deal multiple error handling blocks at a time like below-

Example:

```
try:
  print('i am try block')
  x = int('ak')
except SyntaxError:
  print(' i am a syntax error')
except (TypeError, ZeroDivisionError) :
  print(' i am a TypeError error or')
  print(' i am a ZeroDivisionError error')
except ValueError:
  print(' i am a Value error')

# Output
i am try block
i am a TypeError error or
i am a ZeroDivisionError error
```

Example:

Let's assume that there is no file called dummy.txt to open

```
with open('dummy.txt') as f:
    f.read()

---------------------------------------------------------------------
---
FileNotFoundError                         Traceback (most recent
call last)
<ipython-input-26-15b580b26aa1> in <module>()
----> 1 with open('dummy.txt') as f:
      2     f.read()

FileNotFoundError: [Errno 2] No such file or directory:
'dummy.txt'
```

The above error situation can be handled like below-

```
try:
  f = open('dummy.txt')
except FileNotFoundError:
  print('sorry that file you are looking for doesnt exist')
else:
  print(f.read())
  f.close()
finally:
  print('the task is done')

#Output
sorry that file you are looking for doesnt exist
```

Errors can have aliases and they can be printed.
Multiple except blocks can also be allowed to deal with multiple errors.
Example:

```
try:
 with open('dummy2.txt','r') as f:
  print(f.readlines())
except ValueError as v:
 print(v)
except NameError as n:
 print(n)
except ZeroDivisionError as z:
 print(z)
```

```
except TypeError as t :
 print(t)
except FileNotFoundError as g :
 print(g)
else:
 print('i am else')
finally:
 print('i am finally')
```

raise keyword:
Exceptions are generated in Python programming when runtime issues occur.
The raise keyword can also be used to manually raise exceptions.

We may optionally give values to the exception to provide more information about why it was raised.
Example:

```
try:
 a = int(input("Enter an integer: "))
 if (a%2) != 0:
   raise ValueError("That is not an even number!")
except ValueError as v:
 print(v)
else:
 print(a,'is an even number')
```

Its totally our choice how to deal with exception handlings, whether to split up the problem statement in the blocks or not.
Its always a good practice to split the task into multiple blocks and dealing possible errors using except block and making use of optional else and finally blocks. It helps in debugging the errors in quicker way and resolve them and also as a clean code practice.

Dates and Time:

date or **time** in python can be used by using a module named **'datetime'.** It's not a data type unlike
other programming languages.

How to display the current date in python?
Example:

```
import datetime
```

```
x = datetime.datetime.now()
print(x)

#OUTPUT : 2021-03-31 13:21:25.062264
```

So, the above output seems to have a lot of stuff in it. Let's see one by one.

The date contains **year, month, day, hour, minute, second,** and **microsecond** respectively**.**

If you want to access any specific thing from the above output, here we go-

Example:

```
import datetime
x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))

'''OUTPUT:
2021
Wednesday
'''
```

Creating Date Objects:

To create a date, we can use the **datetime( )** class (constructor) of the datetime module.

The **datetime( )** class requires three parameters to create a **date: year, month, day**.

Example:

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)

#OUTPUT : 2020-05-17 00:00:00
```

A Reference for legal date format codes-

Python Date Format Codes

Python Library Vs Package Vs Modules-

A **library** is a catch-all phrase for a reusable piece of code. A Python library typically consists of a collection of related modules and packages.
It's a reusable block of code that we may utilize in our application by importing it and invoking the library's method with a period (.)

The **package** is just a directory with a bunch of modules in it. This directory includes Python modules as well as a init .py file that the interpreter uses to recognise it as a Package. The package is nothing more than a namespace. Within the package, there are sub-packages.

The **module** is a basic Python file with a.py extension that holds collections of functions and global variables. It is an executable file, and the idea of Package in Python is used to arrange all of the modules.

However, it is commonly understood that a library is a collection of packages, whereas a package is a collection of modules.

Python Modules

- os module
- math module
- statistics module
- random module
- date time module

os module

In Python, the OS module has functions for users to interact with the operating system. Python's basic utility modules include OS. This module allows you to use operating system-dependent functions on the go.

How to import os module and print current working directory?

```
import os #import is a keyword, os is a module name
os.getcwd() #get current working directory

#Output
/content  #prints current working directory
```

How to list all the contents of the directory?

```
os.listdir('/content')

#Output
```

```
['.config', 'sample_data'] #lists all the folders and files of the given directory
```

Note: Files or Folders that start with period(.) as first character are Hidden documents. Hidden files or folders are not visible usually in any GUI.

How to change operation from one directory to another directory?

```
os.getcwd()

#output
/content
```

```
os.chdir('/content/akhil') #changes directory to akhil
```

```
os.getcwd()

#output
/content/akhil
```

How to create a directory?

```
os.mkdir('akhil') #make or create your own directory or folder
```

```
os.listdir('/content')

#Output
['.config', 'sample_data', 'akhil'] #lists all the folders and files of the given directory
```

How to remove a particular directory?

Note: You cannot remove the current working directory, for that you have to position yourselves out of that directory.

Let's assume that you are already in **'akhil'** folder and try removing it.

```
os.rmdir('/content/akhil') # Tries to Remove the specified directory
# Note: You cannot remove current working directory
```

```
#Output
No such file or directory: 'akhil'
```

```
os.chdir('/content') #changes directory to content
```

```
os.getcwd()

#output
/content
```

```
os.rmdir('/content/akhil') # Remove the specified directory
```

How to rename a directory?

os.rename(‘oldname’, ‘newname’)

```
os.rename('akhil', 'abc') #renames directory akhil to abc
```

```
os.listdir('/content')

#Output
['.config', 'sample_data', 'abc'] #lists all the folders and files
of the given directory
```

Math module

Python has built-in math module for all mathematical calculations.

```
import math
print(math.pow(2,3)) #2**3
print(math.sqrt(64))

#Output
8
8
```

```
math.floor(2.3) #Rounds off value to previous decimal, i.e.,2
math.ceil(2.5) #Rounds off value to next decimal i.e., 3

#Output
2
3
```

```
print(math.pi)
print(math.e)

#Output
3.14285
2.718281828459045
```

```
print(math.log(10))
print(math.log10(10))

#Output
2.302585092994046
1.0
```

```
print(math.factorial(4))

#Output
24
```


Statistics module

Python has built-in statistics module for all statistical calculations.

```
import statistics
statistics.mean([1,2,3,4,5]) #15/5 = 3.0
statistics.median([1,2,3,4,5]) #middle value = 3
statistics.median([1,2,3,4,5,6]) # middle values average=(3+4)/2 =
3.5
statistics.mode([1,22,33,4,45,5,6,6,66]) #most repeated value
statistics.harmonic_mean([1,2,3,4,5]) #2.189
```

Random module

Python has built-in random module for random number operations.

```
import random
random.random()  # returns a random float value between 0 - 1
random.randint(1,101) # returns a random int value between 1 - 100
random.randrange(0,11,2) #returns an even int value between 0 - 10
random.randrange(0,11,2) #returns an odd int value between 0 - 10
random.uniform(1,100) # returns a random float value between 1 -
100
```

```
random.choice('python') #randomly returns a character of string
random.choice([1,2,3,4,5])#randomly returns an item of the list
```

```
a = [1,2,3,4,5]
random.shuffle(a) #Shuffles the order of the items of the list.
print(a)

#Output
[5, 2, 3, 1, 4]
```

seed():

The random number generator is initialised using the **seed()** function.

To create a random number, the random number generator requires a starting value (seed value).

```
random.seed(1)
print(random.random())
```

The generator generates a random number depending on the seed value, therefore if the seed value is 1, the first random number will always be same generated number

```
random.seed(1)
print(random.random())
```

Datetime module

Python has built-in date module for all datetime operations.

- datetime class
- date class

- time class
- time delta class

```
import datetime
d = datetime.datetime.now()
print(d)
```

When you use now() on datetime, it returns output the below format:

year, month, day, hour, minute, second, and microsecond

```
import datetime as dt
dt.datetime.now()
```

timestamp() method returns datetime output in micro seconds format

```
print(d.timestamp())
```

Fixing customised datetime:

```
datetime.datetime(2021, 8, 25, 16, 22, 52, 861560)
```

Date module-

```
from datetime import date
z = x.date.today()
print(z)
print(z.year)
print(z.month)
print(z.day)
```

You can customize the date like below-

```
z = x.date(2020,8,18) #customised date
print(z)
```

Time module-

Customizing the time can be done like below-

```
from datetime import time
```

```
a = time(hour= 20,minute = 30, second = 50)
print(a.hour)
print(a.minute)
print(a.second)

#Output
20
30
50
```

Timedelta

Difference between two timeframes can be calculated like below-

```
from datetime import timedelta
a = timedelta(hours= 20,minutes = 30, seconds = 50)
b = timedelta(hours= 10,minutes = 10, seconds = 10)
print(a-b)

#Output
10:20:40
```

```
from datetime import timedelta
a = timedelta(weeks=10,hours= 20,minutes = 30, seconds = 50)
b = timedelta(weeks =6,hours= 10,minutes = 10, seconds = 10)
print(a-b)

#Output
28 days, 10:20:40
```

```
t1 = datetime.timedelta(weeks = 4)
t2 = datetime.timedelta(weeks = 2)
print(t1-t2)

#Output
2 weeks
```

Searching Techniques:

1. Linear Search
2. Binary Search

Linear Search:

Linear search is a search that searches an element in a list by searching the elements in the list consecutively until the element is found.

Sorting the items of the iterable is essentially not necessary.

```
num = 99999
a = [i for i in range(1000000)]
def linear(a,num):
  for i in a:
    if i==num:
      print(a.index(i))
linear(a,num)

#Output
99999
```

Time taken to execute above linear search can be found out using builtin **timeit()** method by passing function name as input parameter.

```
%timeit(linear)

#Output
The slowest run took 45.93 times longer than the fastest. This
could mean that an intermediate result is being cached.
10000000 loops, best of 5: 35 ns per loop
```

The linear search is not an efficient approach for the huge data.

For example,

For 10 items at the worst case it takes 10 iterations to find the searching element.

For 100 items at the worst case it takes 100 iterations to find the searching element.

For 1000 items at the worst case it takes 1000 iterations to find the searching element.

For 10000 items at the worst case it takes 10000 iterations to find the searching element.

And so on.

So, as its totally dependent on number of items which is a topic of Time Complexity and Big O Notation which is out of the scope of this course. Hence, we have a better approach than linear search which is a Binary Search.

Binary Search:

A binary search is one that recursively searches the middle element in a list until the middle element matches a searched element.

Algorithmic Steps for doing the binary search:

- Items have to sorted either in ascending order or Descending order.
- Calculate the middle item of the list.
- If the searched value is lower than the middle value, set a new right bounder.
- If the searched value is higher than the middle value, set a new left bounder.
- If the search value is equal to the value in the middle value, return the middle index.
- Repeat the steps above until the value is found or the left bounder is equal or higher the right bounder.


```
# Binary Search
num = 99999
a = [i for i in range(1000000)]
def Binary_search(a,num):
  start=0
  end=len(a)-1
  mid = 0
  while (start<=end):
    mid_ind = (start+end)//2
    # print(mid_ind)
    mid_val = a[mid_ind]

    if (num == mid_val):
      return mid_ind

    elif (num<mid_val):
      end = mid_ind-1

    else:
```

```
      start = mid_ind+1
  return 'not exist'
Binary_search(a,num)

#Output
99999
```

Time taken for Binary search:

```
%timeit(Binary_search)

#Output
The slowest run took 45.78 times longer than the fastest. This
could mean that an intermediate result is being cached.
10000000 loops, best of 5: 33.7 ns per loop
```

Object Oriented Programming System (OOPS)

Object Oriented Programming is a programming paradigm based on the concepts of 'Object' which contains data in the form of fields(variables,functions,properties..)

- Object:
    - Attributes or properties
        - Name, Height, weight, color
    - Behavior or functionality
        - Walking, running, thinking

Example: A Human is Object,

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

```
a = 3
print(type(a))

#Output
<class 'int'>
```

As we can see that the output of the above code gives class int which proves that everything that python provides for us is essentially a class behind the screens.

Basic Principles of OOPS are as follows:

Class:

- A class is a Blueprint for the object.
- A collection of objects can be specified as a class.
- It's a logical entity with certain unique properties(attributes) and functions(methods).

Note: Naming convention suggests that class names should always be **Capitalized** in order to avoid conflict with method names.

```
class Human:
  pass
```

Object:

- The object is a self-contained entity with state and behavior. It might be anything that exists in the actual world.
- All real world entities are called 'Object' . Ex: Car, bus, building, human
- An object is an instantiation of a class containing attributes or properties or methods.

A class shall generate an object to allocate memory when it is defined.

```
obj = Human() #Object declaration/Instantiation
Obj

#Output
<__main__.Human at 0x7fa3015bf310>
```

Creating attributes to object "**obj**"-

```
obj.name = 'ironman'
obj.name

#Output
ironman
```

```
obj.height = 4
obj.height

#Output
4
```

Creating attributes to object "**obj2**"

```
obj2 = Human()
obj2.NAME = 'superman'
obj.height = 40
```

List of methods and attributes of an object can be seen using **dir()** method-

```
dir(Human)
```

if you don't fix attributes, any number of attributes can be created by objects or developers which is a bad approach

```
class Human:
  pass

human1 = Human()
human2 = Human()

human1.name = 'Akhil'
human2.Name = 'Teja'

print(human2.name) #ERROR
print(human1.Name) #ERROR
print(human1.NAME) #ERROR
print(human2.Name) #Teja

#Errors occurred because attribute names are case sensitive, so we need to set a standard attribute for everybody to follow
```


What is a Constructor?

A Constructor is a unique method for creating and initializing a class object. A destructor, on the other hand, is used to delete the object.This method is defined in the class.

- The constructor is called automatically when an object is created.
- A constructor's main purpose is to define and initialize a class's data member/instance variables. The constructor is a set of statements (or instructions) that run when an object is created and are used to set up the object's characteristics.

In Python, we have the following three types of constructors.

- Default Constructor
- Non-parameterized constructor
- Parameterized constructor

Example:

```
def __init__(self):
    # body of the constructor
```

self keyword:

- The first argument self refers to the current object. It binds the instance to the __init__() method. It's usually named self to follow the naming convention.
- In the method definition, class methods must have an additional initial argument. When we call the method, we don't specify a value for this parameter; Python does.
- We still need one parameter if we have a method that doesn't take any arguments.
- The first parameter does not have to be called self. We may give the method whatever name we like, but it must be the first parameter.

```
class Human:
  # class attribute
  name = 'Ironman'

  # method
  def walking(self):
# self is an implicit instance/ implicit reference, it always points to the current object
   # self is like a parameter to a function, acts like a catalyst and calls the object.
   # self must be first parameter
    print('Ironman is walking')

h1 = Human()

h1.walking()

#Output
Ironman is walking
```

Default Constructor:
Python will provide a default constructor if no constructor is defined.

```
class Human:
```

```
    def display(self):
        print('Human Display')

he = Human()
he.display()
```

__init__() :

The **__init__** method is called a constructor.
When a class object is instantiated, it executes this code. Any initialization you wish to do with your object may be done with this function.

Non-Parameterized Constructor:
A constructor without any arguments is called a non-parameterized constructor.

```
class Car:
  # class attribute
  name = 'Tata'

  # constructor
  # Instance attribute
  def __init__(self): #self is an implicit instance. It points
current instance
    print('hi',self.name)

car1 = Car()

# Output
hi Tata
```

Parameterized Constructor:
A constructor with defined parameters or arguments is called a parameterized constructor.
If name is taken as an input-

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

  def walking(self):
    print(f'{self.name} is walking')
```

```
h1 = Human(‘akhil’)

h1.walking()

#Output
Ironman is walking
```

Example on how list data type works as a class:

```
class list:
  # class attribute
  name = 'Ironman'

  # method
  def append(self,val):
    a += [val]

a = list()

a.append(3)
```

Example:

```
class Car:
  # class attribute
  wheels = 4

  # Instance attribute
  def __init__(self,engine,speed,color): #self is an implicit
instance. It points current instance
    self.engine = engine
    self.speed = speed
    self.color = color

Tesla = Car('electric',150,'black')
Benz = Car('petrol',100,'blue')
Audi = Car('diesel',120,'red')
```

```
print(Tesla.speed)
print(Benz.engine)
print(Audi.color)
```

```
#output
150
petrol
red
```

NOTE:

- The constructor will be called just once for each object. The constructor is called four times if we create four objects, for example.
- Every class in Python has a constructor, although it isn't necessary to declare it explicitly. It's not required to define constructors in a class.
- If no constructor is specified, Python will provide a default constructor.

OOPS Variables:

There are two types of variables:

- Class variables
    - A Python class variable is shared by all object instances of a class. Class variables are declared when a class is being constructed. They are not defined inside any methods of a class.

- Instance variables
    - A class method declares instance variables. An instance variable's value can vary based on the instance with which it is connected.
    - This means that any instance variable's value can be varied. This differs from a class variable, which can only have one value.

name acts as both class variable and instance variable whereas
height, weight are instance variables in this example-

```
# Class variable Vs Instance variable
class Human:
 # Class variable shared by all objects
 name = 'Ironman'

 # Instance variable shared by specific objects and can be changed
 def __init__(self,name,height,weight):
   # name = 'akhil'
   self.name = name
   self.height = height
   self.weight = weight
```

```
 def walking(self): #here self can be object-a or object-b
   print(f'{self.name} is walking')

a = Human('spiderman',5,40)
b = Human('superman',6,50)

a.name
b.name
b.name='batman' #will override class variable name
b.name #updates previous name to batman in this particular
instance only
a.name #same as previous a.name

# if you want to update class variable
Human.name = 'waterman'
Human.name
```

```
class Car:
  # class attribute
  name = 'fancy car'

  # Instance attribute
  def __init__(self,engine,speed,color): #self is an implicit
instance. It points current instance
    self.engine = engine
    self.speed = speed
    self.color = color
    print('hi')

# Tesla = Car('electric',150,'black')
# Benz = Car('petrol',100,'blue')
Audi = Car('diesel',120,'red')

# Output
hi
```

There are three types of methods:

- Instance method:
  - If you want to modify on **instance variable**
  - **No** decorator needed

    - **self** as first argument
- Class method:
    - Class methods are for when you need to have methods that aren't specific to any particular instance, but still involve the class in some way.
    - If you want to modify **class variable**
    - Decorator needed
    - **cls** as first argument
- Static method
    - Decorator needed
    - **No** arguments needed

```
# Class method, instance method, static method
class Human:
 # Class variable shared by all objects
 name = 'Ironman'

 # Instance variable shared by specific objects
 def __init__(self,name,height,weight):
   self.name = name
   self.height = height
   self.weight = weight

 # Instance method should have self
 def walking(self): #here self can be object-a or object-b
   print(f'{self.name} is walking')

 # class method should have decorator
 @classmethod
 def updatename(cls,name): #Instead of self, we should use cls to
call the object
   cls.name = name
   print(f'Updated name is {cls.name}')

 @staticmethod
 def movie(): # Static method takes NO self NO cls, No arguments
should be passed
   print('I am static method')

a = Human('spiderman',5,40)
b = Human('superman',6,50)

Human.updatename('fireman') #Updates the class variable using
```

```
class method
Human.name
```

Properties of OOPS:

- Inheritance
    - Single
    - Multilevel
    - Hierarchical
    - Multiple
- Polymorphism
    - Overloading(Compile error)
    - Overriding(Run time error)
- Abstraction
- Encapsulation
    - Public
    - Private
    - Protected

Inheritance:

Single Inheritance-

Inheritance refers to a class's ability to derive or inherit properties from another class.

The derived class, also known as a child class, is the one for which properties are derived, whereas the base class, also known as a parent class, is the one from which properties are derived.

All parent class methods and attributes are accessible to child class object. But a parent class object cannot access child class methods or attributes.

```
# Single Inheritance
class Parent:
  a = 2
  b = 3
  def pfun(self):
    print('I am Parent class/ Base class')

class Child(Parent):
  c = 4
  d = 5
```

```
  def cfun(self):
    print('I am child class/derived class')

pobj = Parent()
pobj.pfun() # 'I am Parent class/ Base class'
pobj.a # 2
pobj.d # Error as parent class object cannot access child class
objects or attributes

cobj = Child()
cobj.cfun() # 'I am child class/derived class'
cobj.pfun() # 'I am Parent class/ Base class', because parent
class object can access child class
cobj.b #3
```

Multilevel Inheritance-

Inherited class can be inherited further for another class.
Parent or Base class objects cannot inherit child class methods or attributes whereas derived or child class objects can inherit from the parent or base classes.

```
# Multi-level Inheritance
class GrandParent:
  a = 2
  b = 3
  def gfun(self):
    print('I am GrandParent class/ Base class')

class Parent(GrandParent):
  c = 4
  d = 5
  def pfun(self):
    print('I am Parent class of child and child class of
grandparent')

class Child(Parent):
  e = 6
  f = 7
  def cfun(self):
    print('I am child class/derived class')
```

```
gobj = GrandParent()
gobj.a #2
gobj.c #error because base classes cannot access derived class
attributes

pobj = Parent()
pobj.b # 3
pobj.c # 4

cobj = Child()
cobj.cfun()
cobj.pfun()
cobj.a
cobj.d
```

Multiple Inheritance-

A Class can inherit attributes or methods from multiple classes at a time.

```
# Multiple Inheritance
class Father:
  a = 2
  b = 3
  def ffun(self):
    print('I am Father class/ Base class')
class Mother:
  c = 4
  d = 5
  def mfun(self):
    print('I am Mother class/ Base class')

class Child(Father, Mother):
  e = 6
  f = 7
  def cfun(self):
    print('I am child class/derived class')

fobj = Father()
mobj = Mother()
cobj = Child()
```

```
fobj.a # 2
fobj.mfun() # Error because Father & Mother are two different
classes

mobj.c # 4
mobj.ffun() # Error because Father & Mother are two different
classes

cobj.a # 2
cobj.c # 4
cobj.e # 6
cobj.mfun() # I am Mother class/ Base class
```

Hierarchical Inheritance-

Any number of classes can inherit from the same class.

```
# Hierarchical Inheritance
class Parent:
  a = 2
  b = 3
  def pfun(self):
    print('I am parent class/ Base class')

class Son(Parent):
  c = 4
  d = 5
  def sfun(self):
    print('I am son class/ derived class')

class Daughter(Parent):
  e = 6
  f = 7
  def dfun(self):
    print('I am daughter class/derived class')


pobj = Parent()
sobj = Son()
```

```
dobj = Daughter()
pobj.d # error because its a parent class object
sobj.a # 2
dobj.b # 3
```

__init__() method gets overwritten in case of inheritance.

```
class Person:
  a = 2
  def __init__(self):
    print('hi')

class Student(Person):
  def __init__(self):
    print('hey')

p = Student()

#Output
hey
```

Alternative ways of inheriting properties or methods is using **super()** method.

- **super()** gives you access to methods in a superclass from a subclass that inherits from it at a high level.
- **super()** by itself returns a temporary object of the superclass, which may subsequently be used to invoke the methods of that superclass.
- Using **super()** to call already created methods avoids having to rebuild those methods in your subclass and allows you to swap out superclasses with little code modifications.

```
# Super method to inherit properties
class Person:
  a = 2
  def __init__(self,firstName, lastName):
    self.firstName = firstName
    self.lastName = lastName

class Student(Person):
  def __init__(self,firstName,lastName,year):
```

```
    super().__init__(firstName,lastName)
    self.year = year

  def welcome(self):
    print(self.year)

d = Student('akhil','teja',2022)
d.welcome()

#Output
2022
```

Polymorphism:

Polymorphism in Python refers to an object's flexibility to take on several forms. To put it simply, polymorphism allows us to do the same activity in a variety of ways.

For example, a female can be a mother, a sister, a daughter, a ceo etc.

You must have used GPS to navigate the route. Isn't it remarkable how many alternative routes you can find for the same destination depending on traffic? This is known as 'polymorphism' in programming. It's an example of an OOP methodology in which a single task can be completed in a variety of ways. To put it another way, it's a property of an item that enables it to take on multiple forms.

Poly refers to Many, Morph refers to Forms

- Compile-time Polymorphism
- Run-time Polymorphism

Compile-time Polymorphism:
A compile-time polymorphism, also known as a static polymorphism, is one that is resolved during the program's compilation. "Method overloading" is a classic example.

Method overloading :
Method overloading is the process of invoking the same method with various arguments. Method overloading is not supported in Python. Even if you overload a method, Python only evaluates the most recently specified version. If you overload a method in Python, you'll get a TypeError.
Method Overloading:

```
def addition(a, b):
    c = a + b
```

```
    print(c)


def addition(a, b, c):
    d = a + b + c
    print(d)


# the below line shows an error
# addition(2, 3)

# This line will call the second product method
addition(2, 3, 4)
```

Default arguments concept can be used for overcoming above errors.

```
class Dummy:

  def add(self,t1,t2=2,t3=3):
    return t1+t2+t3

d = Dummy()
# d.add(10)
# d.add(10,20)
d.add(10,20,30)
```

Operator Overloading:
Operator overloading refers to modifying an operator's default behaviour based on the operands (values) we use. In other words, the same operator can be used for many purposes.

Here Same operator “plus” is used for adding and also concatenation.

```
a,b,c = 1,2,3
a+b+c
#6

a,b,c = '1','2','3'
a+b+c
#123
```

Functions and Objects Polymorphism:

Polymorphism may be achieved by writing a function that accepts any object as an argument and executes its method without verifying its class type. Instead of repeated method calls, we may call object actions with the same function.

Example-2:

```
class Dog:
  def Sound(self):
    print('bow')

class Cat:
  def Sound(self):
    print('Meow')

def makesound(animal):
  return animal.Sound()

cat = Cat()
dog = Dog()
makesound(dog)
```

Method Overriding:

We can define methods in the child class that have the same name as the methods in the parent class using method overriding polymorphism. Method Overriding refers to the practise of reimplementing an inherited method in a child class.

Example:

```
# Method Overriding
class Bike1:
  def Gears(self):
    return 3

class Bike2(Bike1):
  def Gears(self):
    return 4

b = Bike2()
b.Gears()
```

Example:

```
class Vehicle:

    def __init__(self, name, color, engine):
        self.name = name
        self.color = color
        self.engine = engine

    def show(self):
        print('Details:', self.name, self.color, self.engine)

    def max_speed(self):
        print('Vehicle max speed is 150')

    def gear(self):
        print('Vehicle has 5 gears')


# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')

    def gear(self):
        print('Car has 7 gears')


# Car Object
car = Car('Car x1', 'Red', 'petrol')
car.show()
# calls methods from Car class
car.max_speed()
car.gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white','diesel')
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.gear()
```


Example:

```
# Variable Overriding
class Man:
  name = 'Pratheek'

class Men(Man):
  name = 'handsome'

m = Men()
m.name
```

Abstraction:

Hiding Implementation details is called as Abstraction
In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency.
Hide Implementation details, returns using call backs

- **ABSTRACT CLASS :** If it has one or more abstract methods
- **ABSTRACT METHOD :** Only declaration, but no Definition
- **CONCRETE CLASS :** If it doesn't have abstract class
- **CONCRETE METHOD :** If it has both definition and declaration

Object cant be instantiated for abstract class
Object can only be instantiated by concrete class

```
# Decorators:
********************
  # finally
********************
```

```
def first(m):
  print(m)

# first('Hello')
second = first
second('Hi')

Hi
```

```
def greet():
  print('Hello Welcome')
```

```
def greet2():
  print('welcome to python')

def dec(fun):
  print('**************')
  fun()
  print('**************')

dec(greet2)
```

```
**************
welcome to python
**************
```

```
def mul1(n):
  def mul(x):
    return x*n
  return mul

t1 = mul1(4)
# t1(3) #mul1(4)(3)

t2 = mul1(5)
# t2(2)

t1(t2(4))
# t2(4) = mul1(5)(4) = 20
# t1(20) = mul1(4)(20) = 80
```

```
def dec(fun):
  def wrapper():
    print('**********')
    fun()
    print('**********')
  return wrapper # give call back if not using @dec

@dec #is same as doing d = dec(greet)
def greet():
  print('WElcome to PYthon')
```

```
# d = dec(greet)

greet()


**********
WElcome to PYthon
**********
```

```
@dec
def greet2():
  print('hi')

greet2()

**********
hi
**********
```

```
def dec1(fun):
  def wrapper():
    print('**************')
    fun()
    print('**************')
  return wrapper

def dec2(fun):
  def wrapper():
    print('-------------')
    fun()
    print('-------------')
  return wrapper

@dec1
@dec2
def greet():
  print('hi')

greet()
```

```
************
------------
hi
------------
************
```

```
# Import ABC-Abstract base class and abstract method from abc
from abc import ABC, abstractmethod
class A(ABC): #Abstract class
  @abstractmethod
  def display(self): #Abstract Method
    return None


# There should be concrete methods for all abstract methods
class Demo(A): #Concrete class
  def display(self): #Concrete Method
    print('hi')

d = Demo()
d.display()


hi
```

```
# Import ABC-Abstract base class and abstract method from abc
from abc import ABC, abstractmethod
class A(ABC): #Abstract class
  @abstractmethod
  def display(self): #Abstract Method
    return None

  @abstractmethod
  def display2(self): #Abstract Method
    return None

class Demo(A): #Concrete class
  def display(self): #Concrete Method
    print('hi')
```

```
  def display2(self): #Concrete Method
    print('hello')

d = Demo()
d.display2()


hello
```

```
# Import ABC-Abstract base class and abstract method from abc
from abc import ABC, abstractmethod
class A(ABC): #Abstract class
  @abstractmethod
  def display(self): #Abstract Method
    return None

  @abstractmethod
  def display2(self): #Abstract Method
    return None

class Demo(A): #Concrete class
  def display(self): #Concrete Method
    print('hi')

class Demo2(Demo):
  def display2(self): #Concrete Method
    print('hello')

d = Demo2()
d.display()


hi
```


Encapsulation:

In Python, encapsulation refers to the principle of grouping data and methods into a single unit. When you construct a class, for instance, you are implementing encapsulation. Encapsulation is demonstrated by a class, which binds all data members (instance variables) and methods into a single unit.

To Secure the implementation of code and attributes from misusage.

Public:

- Accessible from anywhere

Protected: (_)Single Underscore

- Accessed within the class and also by inherited class

Private: (__)Double Underscore

- Accessible within the class only

Public:

Example1:

```
# Public
class Human:
  a = 'I am public variable'
  def dummy(self):
    print('I am public method')

h = Human()
# h.a
h.dummy()

#Output
I Am a public method
```

Example2:

```
# Public variable example
class Avenger:
 def __init__(self, name, color):
    # public data members
    self.name = name
    self.color = color
    print(f'{self.name} is in {self.color} color ')

 def show(self):
    # accessing public data member inside the class
    print("Name: ", self.name, '; Color:', self.color)
    print(f'{self.name} is in {self.color} color ')
```

```
class Power_ranger(Avenger):
  def show2(self):
     print(f'{self.name} is in {self.color} color ')

a=Avenger('Iron man','red')
print('-'*10)

a.show()
print('-'*10)

# Accessing public data member outside the class
print("Name: ", a.name, 'Color:', a.color)

print('-'*10)
# Accessing public data member in the inherited class
b = Power_ranger('Thor','white')

b.show2()
```

Protected:

Example1:

```
# Protected
class Human:
  _a = 'I am Protected variable'
  def _dummy(self):
    print('I am Protected method')

class Human2(Human):
  def dummy2(self):
    print(self._a)

h = Human2()
# h.dummy2()
# h._a
h._dummy()

#Output
I Am a public method
```

```python
# Private (__)
class Human:
  __a = 'I am Private variable'
  def dummy(self):
    print('I am Private method')
    print(self.__a)

class Human2(Human):
  def dummy2(self):
    print(self.__a) #error

h = Human()
# h.__a #error
# h.dummy()
# h = Human2()
# h.dummy2()
# If you want to access private variables outside the class:
_className__variablename
h._Human__a
```