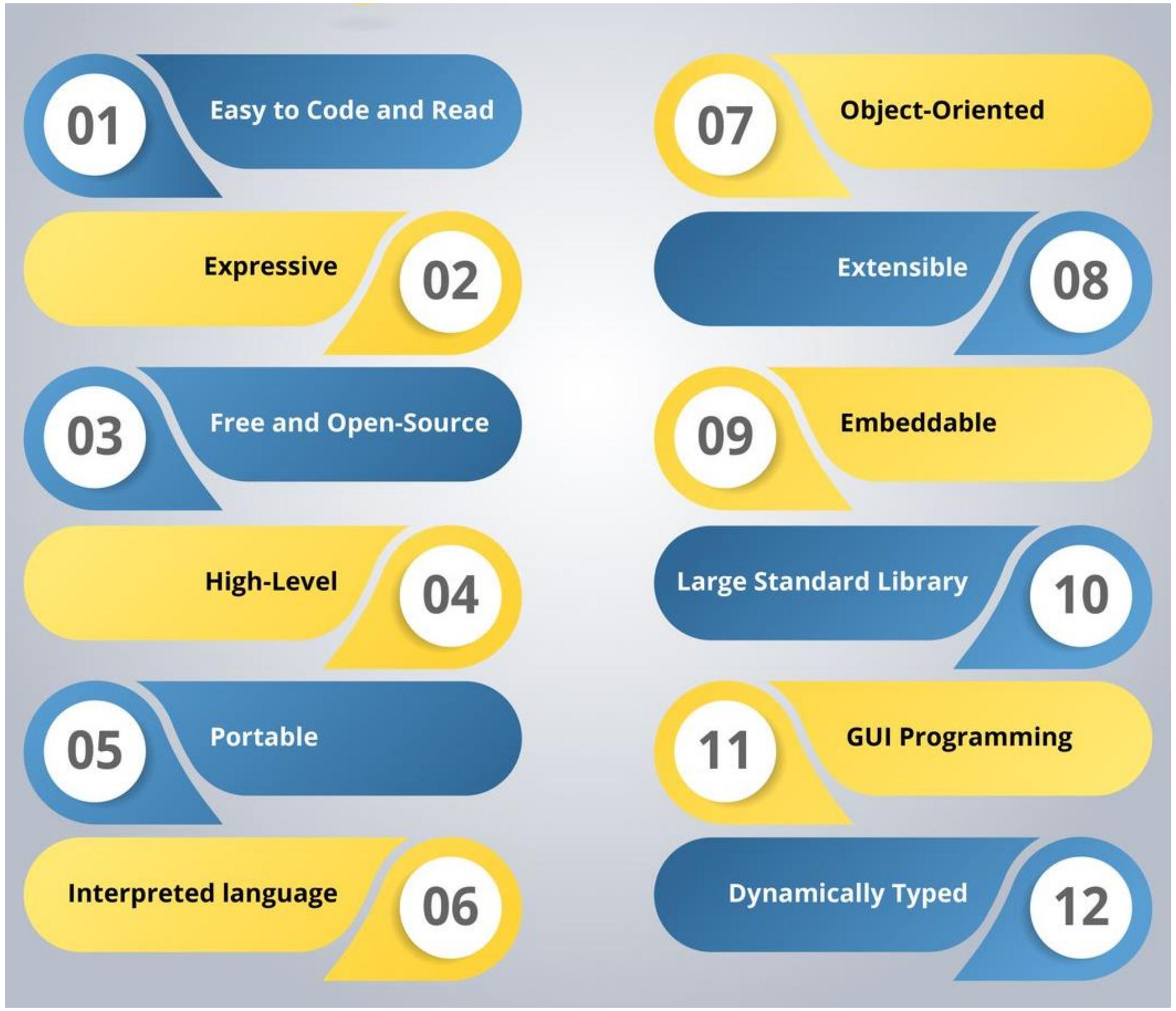
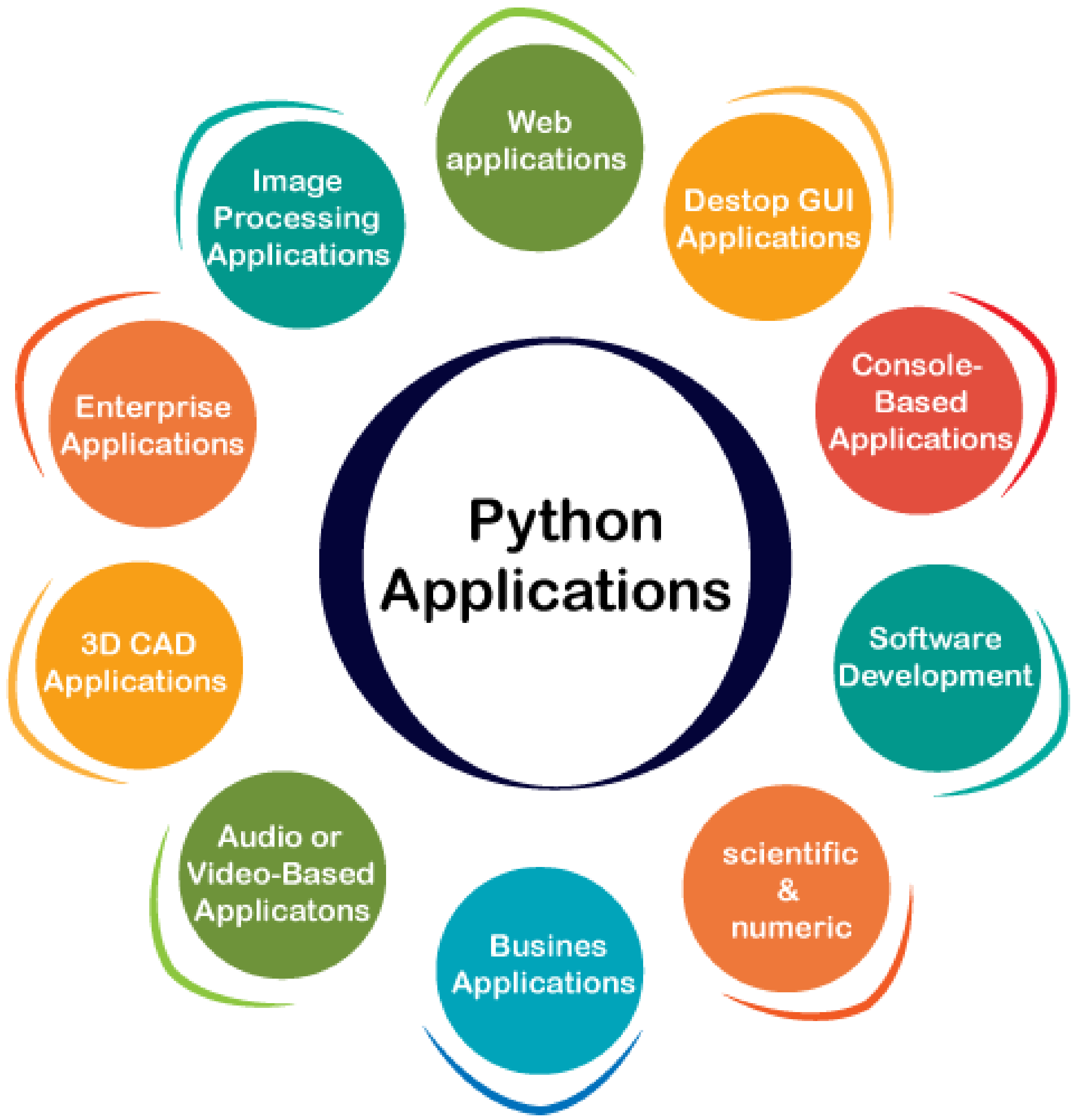
PYTHON

- Python is a cross-platform, general-purpose programming language.
- It is free and open-source.

Python Features:



Python Applications



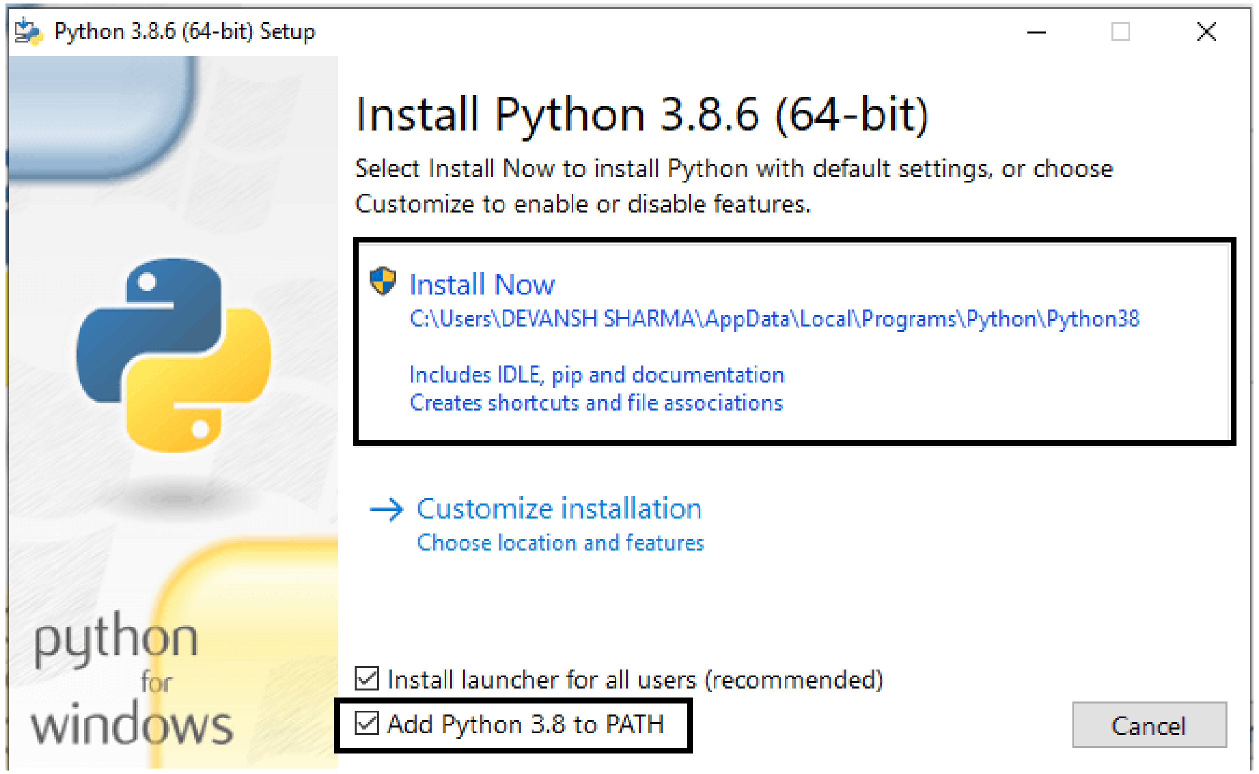
Environment Set-up

Installation on Windows

Visit the link https://www.python.org/downloads/ to download the latest release of Python.

Step - 1: Select the Python's version to download and download it.

Step - 2: Once the file is downloaded, click on the Install Now



Python 3.8.6 (64-bit) Setup

Install Python 3.8.6 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\DEVANSH SHARMA\AppData\Local\Programs\Python\Python38

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.8 to PATH

Now, try to run python on the command prompt. Type the command python --version.

Python Keywords

- Keywords are the reserved words in Python.
- We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.

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

- All the keywords except True, False and None are in lowercase and they must be written as they are.

Python Identifiers

- An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for writing identifiers

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore _. Names like myClass, var_1 and print_this_to_screen, all are valid example.
- An identifier cannot start with a digit. 1variable is invalid, but variable1 is a valid name.
- Keywords cannot be used as identifiers
- We cannot use special symbols like !, @, #, $, % etc. in our identifier.
- An identifier can be of any length.

Things to Remember

- Python is a case-sensitive language. This means, Variable and variable are not the same.
- Always give the identifiers a name that makes sense. While c = 10 is a valid name, writing count = 10 would make more sense, and it would be easier to figure out what it represents when you look at your code after a long gap.
- Multiple words can be separated using an underscore, like this_is_a_long_variable

Python Statement

Instructions that a Python interpreter can execute are called statements. For example, **a = 1** is an assignment statement. if statement, for statement, while statement, etc. are other kinds of statements which will be discussed later.

Multi-line statement

In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\)

```
a = 1 + 2 + 3 + \
    4 + 5 + 6
```

- This is an explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ], and braces { }

```
a = (1 + 2 + 3 +
    4 + 5 + 6 )
```

- Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }

```
colors = ['red',
          'blue',
          'green']
```

- We can also put multiple statements in a single line using semicolons, as follows:

```
a = 1; b = 2; c = 3
```

Python Indentation

- Most of the programming languages like C, C++, and Java use braces { } to define a block of code. Python, however, uses indentation.
- A code block (body of a function, loop, etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.
- Generally, four whitespaces are used for indentation and are preferred over tabs.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

- Indentation can be ignored in line continuation, but it's always a good idea to indent. It makes the code more readable

```
if True:
    print('Hello')
    a = 5
```

```
if True: print('Hello'); a = 5
```

- Both are valid and do the same thing, but the former style is clearer.
- **Incorrect indentation will result in *IndentationError*.**

Python Comments

- In Python, we use the hash (#) symbol to start writing a comment.
- It extends up to the newline character. Comments are for programmers to better understand a program. Python Interpreter ignores comments.

```
#This is a comment
#print out Hello
print('Hello')
```

Multi-line comments

- We can have comments that extend up to multiple lines. One way is to use the hash(#) symbol at the beginning of each line.
- Another way of doing this is to use triple quotes, either ''' or """.

```
#This is a long comment
#and it extends
#to multiple lines
```

```
"""This is also a
perfect example of
multi-line comments"""
```

Python Variables

- A variable is a named location used to store data in the memory. It is helpful to think of variables as a container that holds data that can be changed later in the program.

```
number = 10
```

- You can think of variables as a bag to store books in it and that book can be replaced at any time.

```
number = 10
number = 1.1
```

Assigning multiple values to multiple variables

```
a, b, c = 5, 3.2, "Hello"
```

```
x = y = z = "same"
```

Rules and Naming Convention for Variables and constants

- variable names should have a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_)
- Create a name that makes sense. For example, vowel makes more sense than v
- If you want to create a variable name having two words, use underscore to separate them.
- Use capital letters possible to declare a constant
- Never use special symbols like !, @, #, $, %, etc.
- Don't start a variable name with a digit.

Python Variables

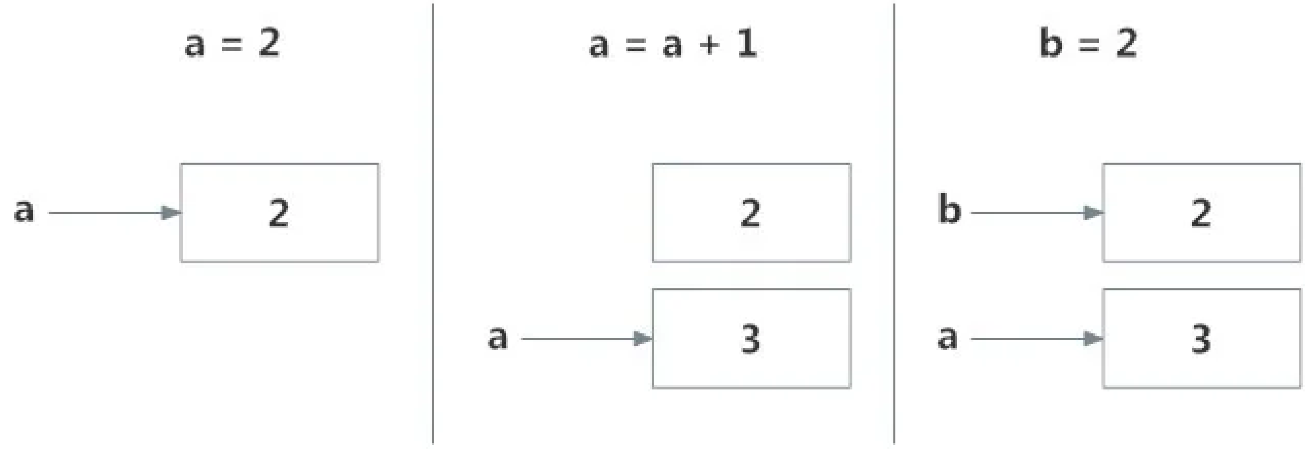
- Name (also called identifier) is simply a name given to objects. Everything in Python is an object. Name is a way to access the underlying object.
- For example, when we do the assignment a = 2, 2 is an object stored in memory and a is the name we associate it with. We can get the address (in RAM) of some object through the built-in function **id().**

```
a = 2
print('id(2) =', id(2))

print('id(a) =', id(a))
```

```
id(2) = 9302208

id(a) = 9302208
```

```
a = 2
print('id(a) =', id(a))

a = a+1
print('id(a) =', id(a))

print('id(3) =', id(3))

b = 2
print('id(b) =', id(b))
print('id(2) =', id(2))
```

```
id(a) = 9302208


id(a) = 9302240

id(3) = 9302240


id(b) = 9302208
id(2) = 9302208
```



Python Literals

- Literal is a raw data given in a variable or constant. In Python, there are various types of literals.

Numeric Literals

- Numeric literals can belong to 3 different numerical types: **Integer**, **Float**, and **Complex**

```
a = 0b111 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal

#Float Literal
float_1 = 10.5
float_2 = 1.5e1

#Complex Literal
x = 3.14j -5

print(a, b, c, d)
print(float_1, float_2)
print(x, x.imag, x.real)
```

```
7 100 200 300
10.5 15.0
(-5+3.14j) 3.14 -5.0
```

Python Literals

String literals

- A string literal is a sequence of characters surrounded by quotes. We can use both single, double, or triple quotes for a string.
- And, a character literal is a single character surrounded by single or double quotes.

```
strings = "This is Python"
char = "C"
multiline_str = """This is a multiline string with
more than one line code."""
unicode = u"\u00dcnic\u00f6de"
raw_str = r"raw \n string"

print(strings)
print(char)
print(multiline_str)
print(unicode)
print(raw_str)
```

```
This is Python
C
This is a multiline string with more than one line code.
Ünicöde
raw \n string
```

Python Literals

Boolean literals

- A Boolean literal can have any of the two values: ***True*** or ***False***.

```
x = (1 == True)
y = (1 == False)
a = True + 4
b = False + 10

print("x is", x)
print("y is", y)
print("a:", a)
print("b:", b)
```

```
x is True
y is False
a: 5
b: 10
```

Special literals

- Python contains one special literal i.e. ***None***. We use it to specify that the field has not been created.

```
a = None
print(a)
print(type(a))
```

```
None
<class 'NoneType'>
```

Python Literals

Literal Collections

- There are four different literal collections **List literals**, **Tuple literals**, **Dict literals**, and **Set literals**.

```
fruits = ["apple", "mango", "orange"] #list
numbers = (1, 2, 3) #tuple
alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary
vowels = {'a', 'e', 'i' , 'o', 'u'} #set

print(fruits)
print(numbers)
print(alphabets)
print(vowels)
```

```
['apple', 'mango', 'orange']
(1, 2, 3)
{'a': 'apple', 'b': 'ball', 'c': 'cat'}
{'a', 'u', 'i', 'e', 'o'}
```

Python Type Conversion and Type Casting

Type Conversion

- 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 Type Conversion
  - Explicit Type Conversion

Implicit Type Conversion

- In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

Converting integer to float

```
num_int = 123
num_flo = 1.23

num_new = num_int + num_flo

print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))

print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

```
datatype of num_int: <class 'int'>
datatype of num_flo: <class 'float'>

Value of num_new: 124.23
datatype of num_new: <class 'float'>
```

- Python always converts smaller data types to larger data types to avoid the loss of data.

Python Type Conversion and Type Casting

Addition of string(higher) data type and integer(lower) datatype

```
num_int = 123
num_str = "456"

print("Data type of num_int:",type(num_int))
print("Data type of num_str:",type(num_str))

print(num_int+num_str)
```

```
Data type of num_int: <class 'int'>
Data type of num_str: <class 'str'>

Traceback (most recent call last):
  File "python", line 7, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Explicit Type Conversion

- In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.
- This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

```
<required_datatype>(expression)
```

- Typecasting can be done by assigning the required data type function to the expression.

Python Type Conversion and Type Casting

Example of Explicit type conversion

Addition of string and integer using explicit conversion

```
num_int = 123
num_str = "456"
print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))
num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))
num_sum = num_int + num_str
print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

```
Data type of num_int: <class 'int'>
Data type of num_str before Type Casting: <class 'str'>

Data type of num_str after Type Casting: <class 'int'>

Sum of num_int and num_str: 579
Data type of the sum: <class 'int'>
```

Key Points to Remember

- Type Conversion is the conversion of object from one data type to another data type.
- Implicit Type Conversion is automatically performed by the Python interpreter.
- Python avoids the loss of data in Implicit Type Conversion.
- Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.
- In Type Casting, loss of data may occur as we enforce the object to a specific data type.

Python Output

- Python provides numerous built-in functions that are readily available to us at the Python prompt.
- Some of the functions like input() and print() are widely used for standard input and output operations respectively

Python Output Using print() function

- We use the print() function to output data to the standard output device (screen)

```
print(*objects, sep=' ', end='\n', file=sys.stdout)
```

- **objects** is the value(s) to be printed.
- The **sep** separator is used between the values. It defaults into a space character.
- After all values are printed, **end** is printed. It defaults into a new line.
- The **file** is the object where the values are printed and its default value is sys.stdout (screen).

```
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4, sep='#', end='&')
```

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

Output formatting

- Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method.

```
print('I love {0} and {1}'.format('bread','butter'))
print('I love {1} and {0}'.format('bread','butter'))
print('Hello {name}, {greeting}'.format(greeting = Welcome, name = 'John'))
```

```
I love bread and butter
I love butter and bread
Hello John, Welcome
```

Python Input

- In Python, we have the input() function to allow this

```
input([prompt])
```

- where prompt is the string we wish to display on the screen. It is optional.
- All the inputs are considered as string. Type casting is required to convert into our desired type.

```
name = input("Enter your name : ")
age = int(input("Enter your name : "))
print("Hi, {0}, your age is {1}".format(name, age))
```

```
Enter your name : kevin
Enter your name : 25
Hi, kevin, your age is 25
```

Python Import

- When our program grows bigger, it is a good idea to break it into different modules.
- A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py.
- Definitions inside a module can be imported to another module
- We use the ***import*** keyword to do this

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

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

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

Python Operators

- Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Arithmetic operators

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

Operator	Meaning	Example
+	Add two operands or unary plus	x + y + 2
-	Subtract right operand from the left or unary minus	x - y - 2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Python Operators

Comparison operators

- Comparison operators are used to compare values. It returns either True or False according to the condition.

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Python Operators

Logical operators

- The logical operators are used primarily in the expression evaluation to make a decision
- Logical operators are the **and, or, not** operators.

Operator	**Meaning**	**Example**
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Python Operators

Bitwise operators

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

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	**Meaning**	**Example**
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

Python Operators

Assignment operators

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

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Python Special Operators

Identity operators

- **is** and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. **Two variables that are equal does not imply that they are identical.**

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

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

print(x1 is not y1)

print(x2 is y2)

print(x3 is y3)
```

```
False

True

False
```

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

Python Operators

- **in** and **not in** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).
- In a dictionary we can only test for presence of key, not the value

Operator	**Meaning**	**Example**
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

```
x = 'Hello world'
y = {1:'a',2:'b'}

print('H' in x)

print('hello' not in x)

print(1 in y)

print('a' in y)
```

```
True

True

True

False
```

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

FLOW CONTROL

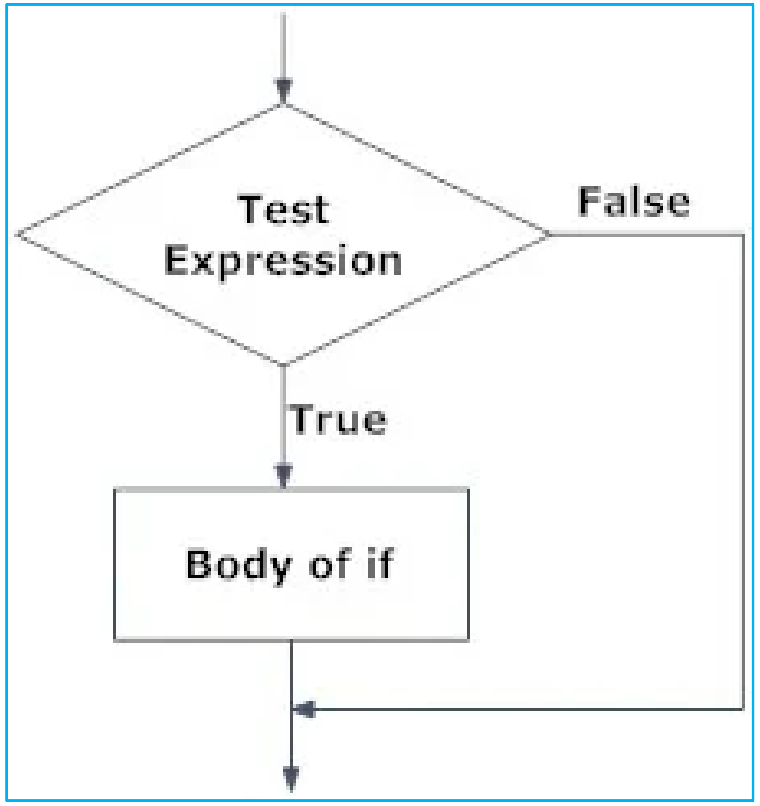
Python if...else Statement

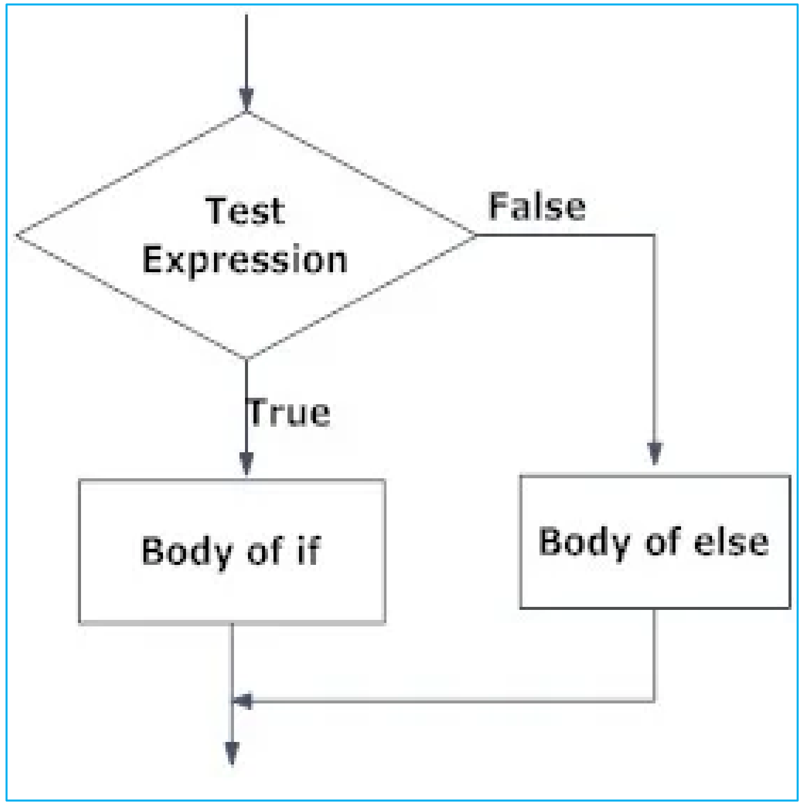
- Decision making is required when we want to execute a code only if a certain condition is satisfied.
- The **if…elif…else** statement is used in Python for decision making.

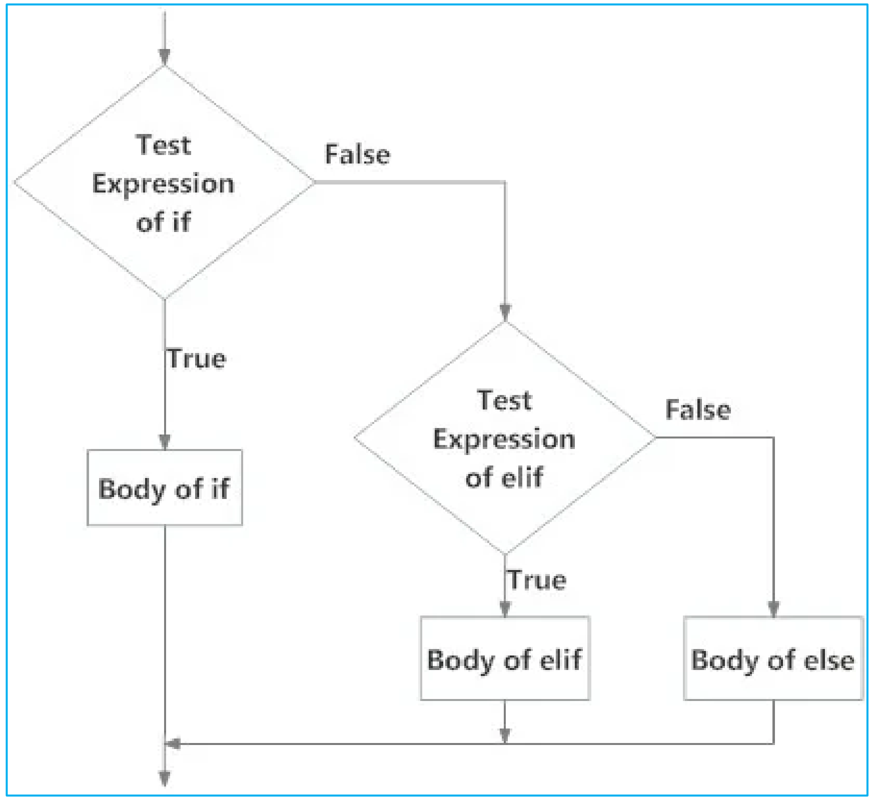
```
if test expression:
    statement(s)
```

```
if test expression:
    Body of if
else:
    Body of else
```

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```







Python if...else Statement

Python Nested if statements

- We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.
- Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

```
num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Case 1:

```
Enter a number: 0
Zero
```

Case 2:

```
Enter a number: 8
Positive number
```

Case 3:

```
Enter a number: -8
Negative number
```

The range() function

- We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as below:

```
range(start, stop,step_size)
```

- To force this function to output all the items, we can use the function list()

```
print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))
```

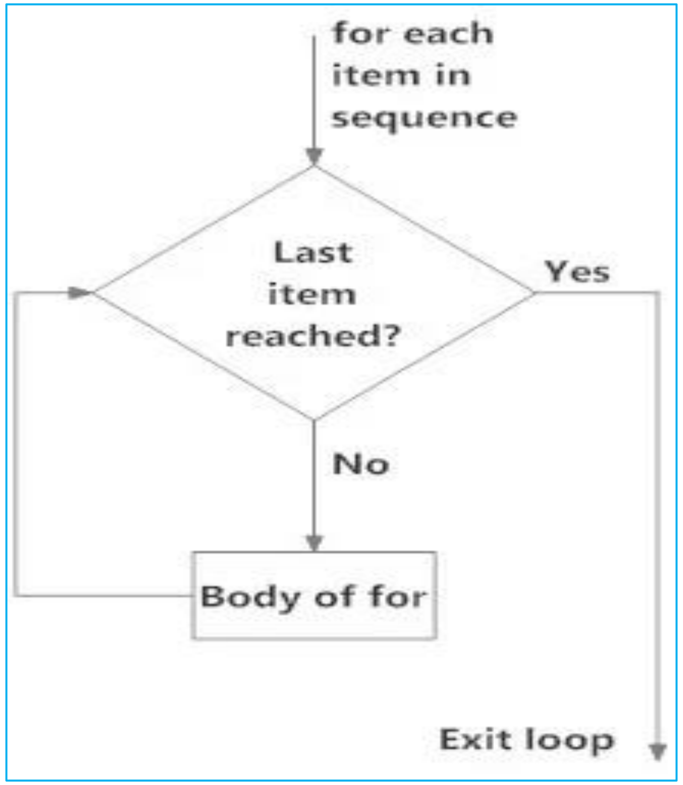
```
range(0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[2, 3, 4, 5, 6, 7]

[2, 5, 8, 11, 14, 17]
```

Python for Loop

- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

```
for val in sequence:
    loop body
```

- Here, val is the variable that takes the value of the item inside the sequence on each iteration.
- Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.



```
numbers = [1, 2, 3, 4, 5]

sum = 0

for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

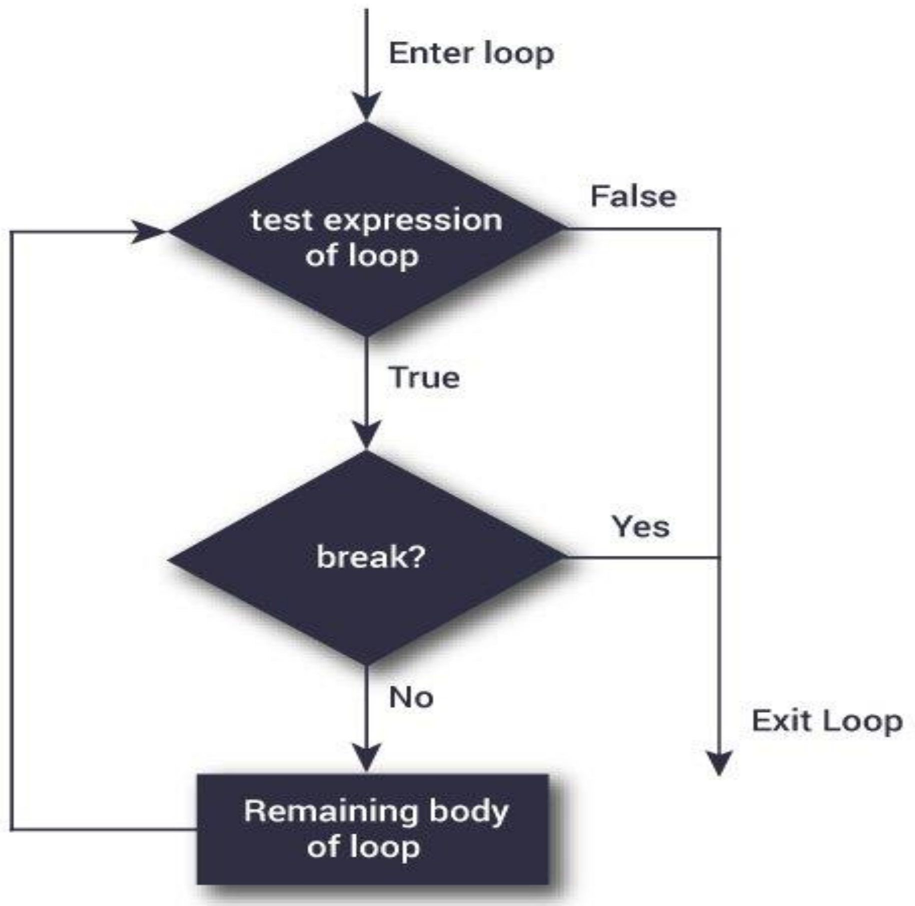
```
The sum is 15
```

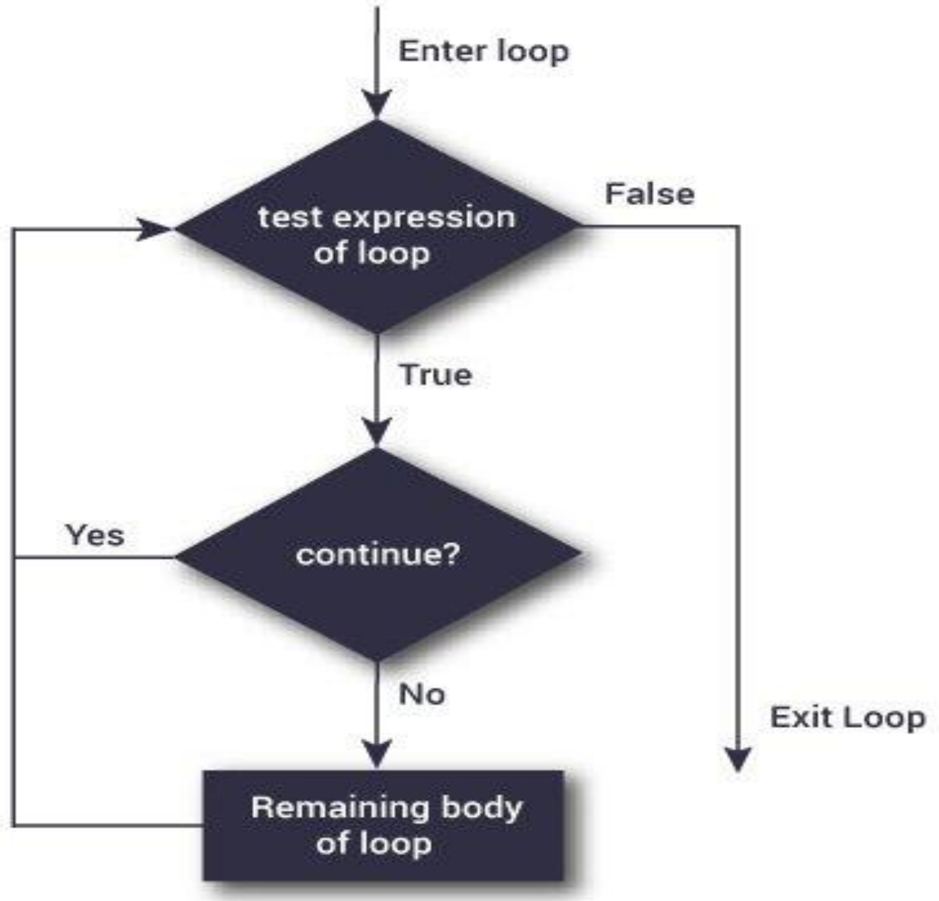
```
sum = 0

for val in range(0,6):
    sum = sum+val

print("The sum is", sum)
```

```
The sum is 15
```

Python break and continue

- In Python, **break** and **continue** statements can alter the flow of a normal loop.
- Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.
- The break and continue statements are used in these cases.





Python for Loop

for loop with else

- A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.
- The break keyword can be used to stop a for loop. In such cases, the else part is ignored.
- Hence, a for loop's else part runs if no break occurs

```
digits = [0, 1, 5]

for i in digits:
    print(i)
else:
    print("No items left.")
```

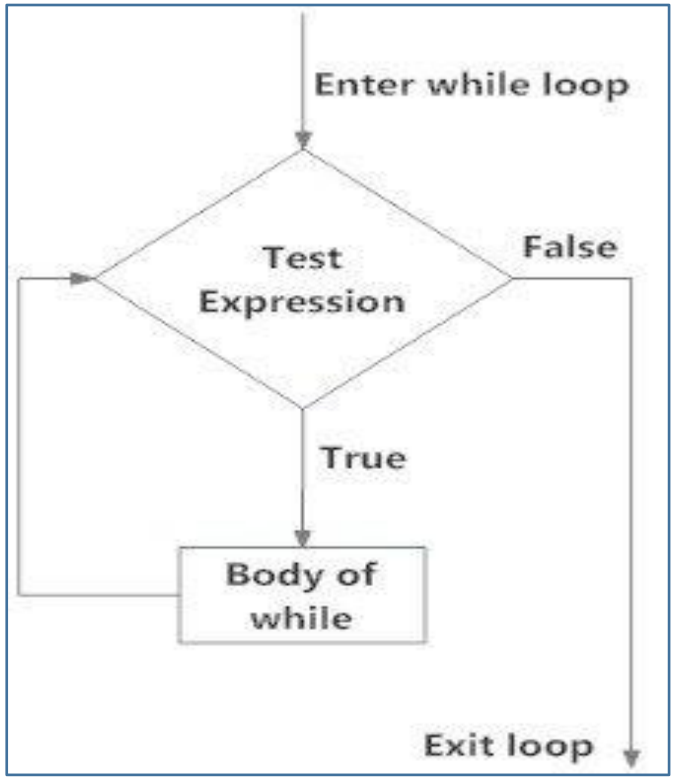
```
0
1
5
No items left.
```

Python WHILE Loop

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- We generally use this loop when we don't know the number of times to iterate beforehand.

```
while test_expression:
    Body of while
```

- In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.



```
n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

```
The sum is 55
```

Python WHILE Loop

While loop with else

- Same as with for loops, while loops can also have an optional else block.
- The else part is executed if the condition in the while loop evaluates to False.
- The while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

```
counter = 0

while counter < 3:
   print("Inside loop")
   counter = counter + 1
else:
   print("Inside else")
```

```
Inside loop
Inside loop
Inside loop
Inside else
```

Functions

Python Functions

- A function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes the code reusable.

```
def function_name(parameters):
            statement(s)
```

```
def greet(name):
    print("Hello, " + name)

greet(“John”)
```

- Keyword def that marks the start of the function header.
- A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
- An optional return statement to return a value from the function.

Note: In python, the function definition should always be present before the function call. Otherwise, we will get an error

Python Functions

The return statement

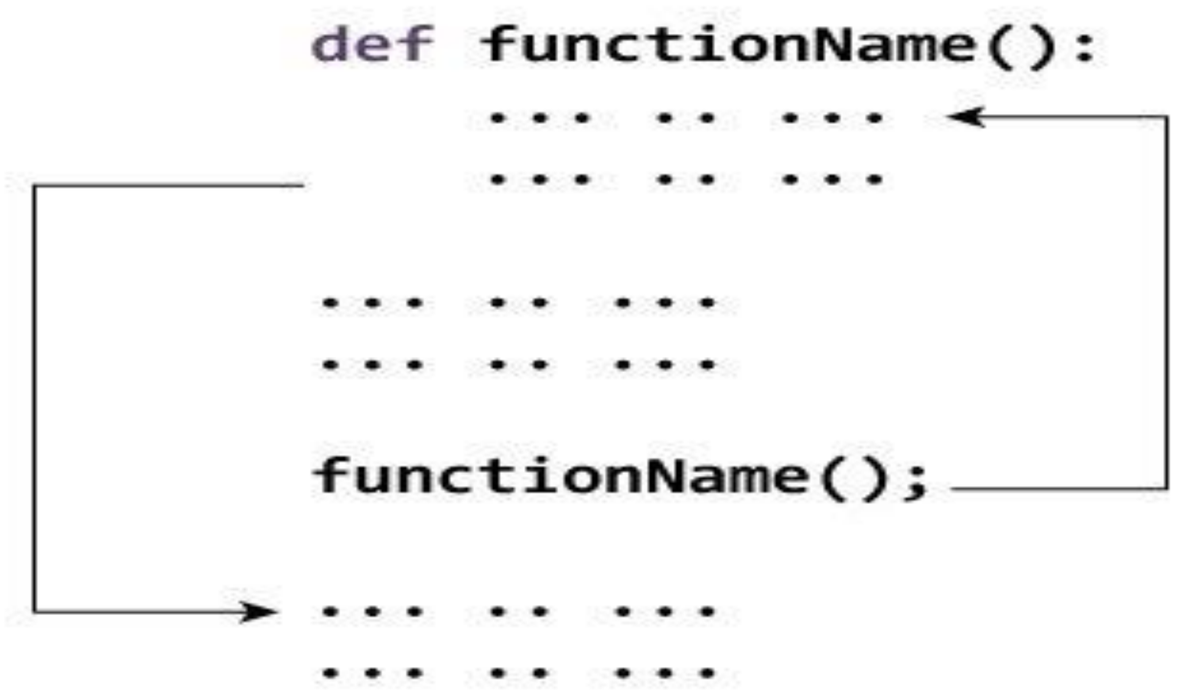
- The return statement is used to exit a function and go back to the place from where it was called.
- Syntax of return

```
return [expression_list]
```

```
def absolute_value(num):

    if num >= 0:
        return num
    else:
        return -num

print(absolute_value(2))

print(absolute_value(-4))
```

```
2
4
```

How Function works in Python?



Scope and Lifetime of variables

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

```
def my_func():
        x = 10
        print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

```
Value inside function: 10
Value outside function: 20
```

```
def my_func():
    global x
    x = 10
    print("Value inside function:", x)

global x
x = 20
my_func()
print("Value outside function:", x)
```

```
Value inside function: 10
Value outside function: 10
```

Function Argument

```
def greet(name, msg):
    print("Hello", name + ', ' + msg)

greet("Peter", "Good morning!")
```

```
Hello Peter, Good morning!
```

Python Default Arguments:

- Function arguments can have default values in Python.

```
def greet(name, msg="Good morning!"):
    print("Hello", name + ', ' + msg)


greet("Kate")
greet("Bruce", "How do you do?")
```

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

- Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

- This means to say, non-default arguments cannot follow default arguments

Function Argument

Keyword Arguments

- When we call a function with some values, these values get assigned to the arguments according to their position.

```
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

- we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.
- Having a positional argument after keyword arguments will result in errors

Function Argument

Arbitrary Arguments

- Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.
- In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument

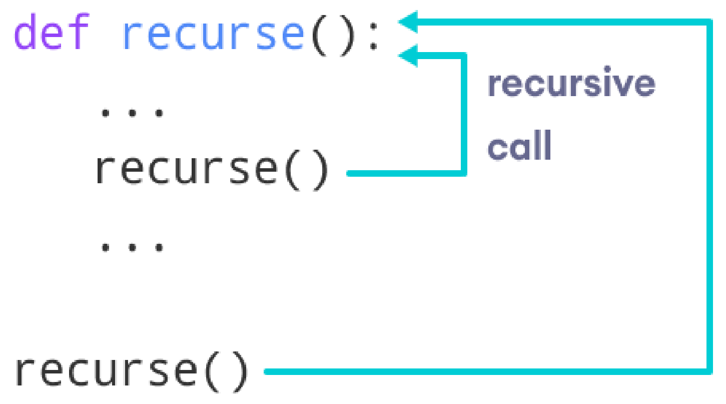
```
def greet(*names):
    for name in names:
        print("Hello", name)


greet("Monica", "Luke", "Steve", "John")
```

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

- Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

What is recursion?

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



```
def recurse():
    ...
    recurse()
    ...

recurse()
```

Datatypes

Datatypes - Numbers

- Python supports integers, floating-point numbers and complex numbers. They are defined as int, float, and complex classes in Python.
- Integers and floating points are separated by the presence or absence of a decimal point. For instance, 5 is an integer whereas 5.0 is a floating-point number.
- Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.
- We can use the type() function to know which class a variable or a value belongs to and isinstance() function to check if it belongs to a particular class.

```
a = 5

print(type(a))

print(type(5.0))

c = 5 + 3j
print(c + 3)

print(isinstance(c, complex))
```

```
<class 'int'>
<class 'float'>
(8+3j)
True
```

Datatypes - LIST

- Python offers a range of compound data types often referred to as sequences. List is one of the most frequently used
- A list is created by placing all the items (elements) inside square brackets [], separated by commas.
- It can have any number of items and they may be of different types (integer, float, string etc.).

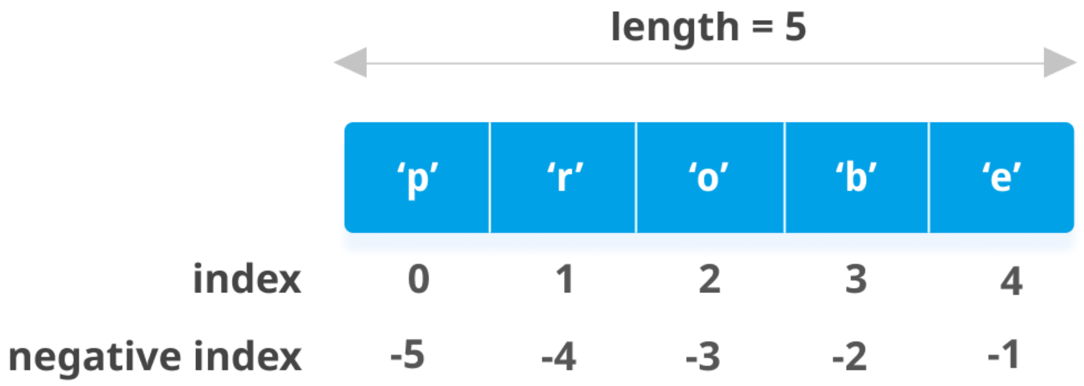
Access List Elements

List Index

- We can use the index operator [] to access an item in a list. In Python, indices start at 0.
- Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError

Negative indexing

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



```
my_list = ['p', 'r', 'o', 'b', 'e']
print(my_list[0])
print(my_list[2])
print(my_list[4])
print(my_list[-1])
print(my_list[-5])
```

```
p
o
e
e
p
```

Datatypes - LIST

How to slice lists in Python?

- We can access a range of items in a list by using the slicing operator :(colon).

```
my_list = ['h','a','s','h','s','k','i','l','l','s']
print(my_list[2:5])
print(my_list[:-6])
print(my_list[4:])
print(my_list[:])
```

```
['s', 'h', 's']
['h', 'a', 's', 'h']
['s', 'k', 'i', 'l', 'l', 's']
['h', 'a', 's', 'h', 's', 'k', 'i', 'l', 'l', 's']
```

Add/Change List Elements

- Lists are mutable, meaning their elements can be changed unlike string or tuple.
- We can use the assignment operator = to change an item or a range of items.
- We can add one item to a list using the append() method or add several items using extend() method.
- we can insert one item at a desired location by using the method insert()
- We can also use + operator to combine two lists. This is also called concatenation.

```
num = [1,2,4]
num[2] = 3
print(num)
num.append(4)
print(num)
num.extend([5,6,7])
print(num)
num.insert(0,0)
print(num)
```

```
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
```

Datatypes - LIST

Delete/Remove List Elements

- We can delete one or more items from a list using the keyword del. It can even delete the list entirely.
- We can use remove() method to remove the given item or pop() method to remove an item at the given index.
- The pop() method removes and returns the last item if the index is not provided.
- We can also use the clear() method to empty a list.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
del my_list[2]
print(my_list)
del my_list[1:5]
print(my_list)
del my_list
print(my_list)
```

```
['p', 'r', 'b', 'l', 'e', 'm']
['p', 'm']
Traceback (most recent call last):
  File "<string>", line 18, in
<module>
NameError: name 'my_list' is not
defined
```

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')
print(my_list)
print(my_list.pop(1))
print(my_list)
print(my_list.pop())
print(my_list)
my_list.clear()
print(my_list)
```

```
['r', 'o', 'b', 'l', 'e', 'm']
o
['r', 'b', 'l', 'e', 'm']
m
['r', 'b', 'l', 'e']
[]
```

Datatypes - TUPLE

- A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas.

A tuple can have any number of items and they may be of different types.

Access Tuple Elements

1. Indexing
2. Negative Indexing
3. Slicing

Changing a Tuple

- Unlike lists, tuples are immutable.
- This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like a list, its nested items can be changed.
- We can also assign a tuple to different values (reassignment)
- We can use + operator to combine two tuples. This is called concatenation.

```
my_tuple = (4, 2, 3, [6, 5])
my_tuple[3][0] = 9
print(my_tuple)
new_tuple = (11,22,33)
my_tuple = my_tuple + new_tuple
print(my_tuple)
my_tuple[1] = 9
```

```
(4, 2, 3, [9, 5])


(4, 2, 3, [9, 5], 11, 22, 33)
TypeError: 'tuple' object
does not support item
assignment
```

Datatypes - TUPLE

Tuple Methods

- Methods that add items or remove items are not available with tuple

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # Output: 2
print(my_tuple.index('l'))  # Output: 3
```

Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Datatypes - SET

- A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
- However, a set itself is mutable. We can add or remove items from it.

```
my_set = set([1, 2, 3, 2])
print(my_set)
```

```
{1, 2, 3}
```

Modifying a set in Python

- Sets are mutable. However, since they are unordered, indexing has no meaning.
- We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.
- We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided

```
my_set = set([1,3])
print(my_set)

my_set.add(2)
print(my_set)

my_set.update([2, 3, 4])
print(my_set)
```

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
```

Datatypes - SET

Removing elements from a set

- A particular item can be removed from a set using the methods discard() and remove().
- The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).
- We can also remove all the items from a set using the clear() method.

```
my_set = {1, 3, 4, 5, 6}
print(my_set)

my_set.discard(2)
print(my_set)

my_set.remove(6)
print(my_set)

my_set.clear()
print(my_set)

my_set.remove(2)
```

```

{1, 3, 4, 5, 6}


{1, 3, 4, 5, 6}


{1, 3, 4, 5}


set()

KeyError: 2
```

Datatypes - DICTIONARY

- Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.
- Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

- Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.
- An item has a key and a corresponding value that is expressed as a pair (key: value).
- While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Accessing Elements from Dictionary

- While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.
- If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
print(my_dict['name'])

my_dict = dict({1:'apple', 2:'ball'})
print(my_dict[1])
```

```
John
apple
```

Datatypes - DICTIONARY

Changing and Adding Dictionary elements

- Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.
- If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

Removing elements from Dictionary

- We can remove a particular item in a dictionary by using the pop() method. This method removes an item with the provided key and returns the value.
- All the items can be removed at once, using the clear() method.
- We can also use the del keyword to remove individual items or the entire dictionary itself.

```
user_dict = dict({"name":"Kevin"})
print(user_dict)

user_dict['age'] = 55
user_dict['city'] = "cbe"
print(user_dict)

user_dict.pop('age')
print(user_dict)

del user_dict['city']
print(user_dict)

del user_dict
print(user_dict)
```

```
{'name': 'Kevin'}

{'name': 'Kevin', 'age': 55, 'city': 'cbe'}

{'name': 'Kevin', 'city': 'cbe'}

{'name': 'Kevin'}

name 'user_dict' is not defined
```

We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.

Python Object Oriented Programming

Object Oriented Programming

- Python is a multi-paradigm programming language. It supports different programming approaches.
- One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  - attributes
  - behavior
- A parrot is an object, as it has the following properties:
  - name, age, color as attributes
  - singing, dancing as behavior
- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

Class

- A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.
- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

```
class ClassName:
   # Statement-1
   .
   .
   .
   # Statement-N
```

Objects

- The object is an entity that has a state and behavior associated with it.

An object consists of :

- **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

The self

1.Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it

2.If we have a method that takes no arguments, then we still have to have one argument.

3.This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

The __init__ method

- The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object

```
class Dog:
    attr1 = "mammal“

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

# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

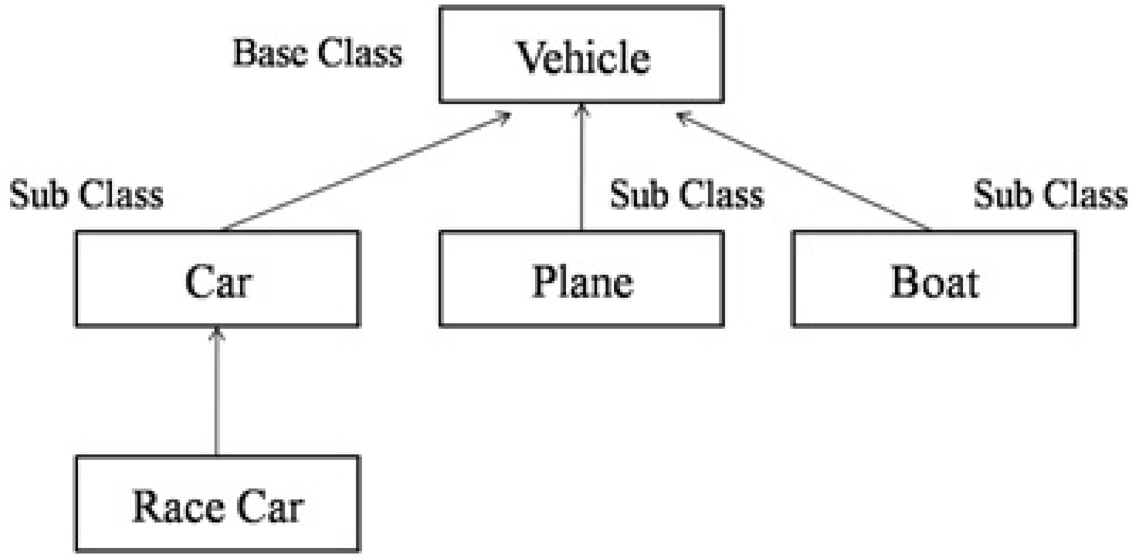
```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Inheritance

- Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class and the class from which the properties are being derived is called the base class or parent class.

The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.



Inheritance - Example

```
# parent class
class Person:
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
```

```
# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))
```

```
# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

```
Rahul
886012
My name is Rahul
IdNumber: 886012
Post: Intern
```

Polymorphism

- Polymorphism simply means having many forms

```
class Bird:

    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
        print("Ostriches cannot fly.")
```

```
obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

```
There are many types of birds.
Most of the birds can fly but some cannot.

There are many types of birds.
Sparrows can fly.

There are many types of birds.
Ostriches cannot fly.
```

Encapsulation

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.

```
# Creating a Base class
class Base:
    def __init__(self):
        self.a = "Hash"
        self.__c = "Skills"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)

obj1 = Base()
print(obj1.a)
```

print(obj1.__c) : 'Base' object has no attribute '__c’
obj2 = Derived() : 'Derived' object has no attribute '_Derived__c'