**Variables**

Variables are containers for storing data values. Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

Rules for Python variables:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _ ).
- Variable names are case-sensitive (age, Age and AGE are three different variables).

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

```
print(myvar)
print(my_var)
print(_my_var)
print(myVar)
print(MYVAR)
print(myvar2)
```

```
John
John
John
John
John
John
```

**Camel Case**

Each word, except the first, starts with a capital letter:

myVariableName = "John"

**Pascal Case**

Each word starts with a capital letter:

MyVariableName = "John"

**Snake Case**

Each word is separated by an underscore character:

my_variable_name = "John"

**Operators**

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

**Arithmetic Operators**

Python Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication, and division.

```
a = 9
b = 4
add = a + b

sub = a - b

mul = a * b

mod = a % b

p = a ** b
print(add)
print(sub)
print(mul)
print(mod)
print(p)
```

```
13
5
36
1
6561
```

**Comparison Operators**

In Python Comparison of Relational operators compares the values. It either returns True or False according to the condition.

```
a = 13
b = 33

print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
print(a <= b)
```

```
False
True
False
True
False
True
```

**Logical Operators**

Python Logical operators perform Logical AND, Logical OR, and Logical NOT operations. It is used to combine conditional statements.

```
a = True
b = False
print(a and b)
print(a or b)
print(not a)
```

```
False
True
False
```

**Bitwise Operators**

Python Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

```
a = 10
b = 4
print(a & b)
print(a | b)
print(~a)
print(a ^ b)
print(a >> 2)
print(a << 2)
```

```
0
14
-11
14
```

```
2
40
```

**Assignment Operators**

Python Assignment operators are used to assign values to the variables.

```
a = 10
b = a
print(b)
b += a
print(b)
b -= a
print(b)
b *= a
print(b)
b <<= a
print(b)
```

```
10
20
10
100
102400
```

**Identity Operators**

In Python, is and is not are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

is : True if the operands are identical

is not : True if the operands are not identical

```
a = 10
b = 20
c = a

print(a is not b)
print(a is c)
```

```
True
True
```

**Data Types**

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are classes and variables are instances (objects) of these classes.

Python has the following data types built-in by default, in these categories:

- Text Type: str
- Numeric Types: int, float, complex
- Sequence Types: list, tuple, range
- Mapping Type: dict
- Set Types: set, frozenset
- Boolean Type: bool
- Binary Types: bytes, bytearray, memoryview
- None Type: NoneType

```
x = "Hello World"
print(type(x))

x = 20
print(type(x))

x = 20.5
print(type(x))

x = 1j
print(type(x))

x = ["apple", "banana", "cherry"]
print(type(x))

x = ("apple", "banana", "cherry")
print(type(x))

x = {"name" : "John", "age" : 36}
print(type(x))


x = {"apple", "banana", "cherry"}
print(type(x))

x = frozenset({"apple", "banana", "cherry"})
print(type(x))

x = True
print(type(x))

x = b"Hello"
print(type(x))

x = None
print(type(x))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
<class 'set'>
<class 'frozenset'>
<class 'bool'>
<class 'bytes'>
<class 'NoneType'>
```

**String**

```
a = "Hello"
print(a)
print(a[1])
print(len(a))

for x in "banana":
  print(x)

# Slicing
b = "Hello, World!"
print(b[2:5])

# Slice From the Start
print(b[:5])

# Slice To the End
print(b[2:])

# Negative Indexing
print(b[-5:-2])

# Python - Modify Strings

a = "Hello, World!"
print(a.upper())
print(a.lower())

# Remove Whitespace
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"

# Replace String
print(a.replace("H", "J"))

# Split String
print(a.split(",")) # returns ['Hello', ' World!']

# String Concatenation
a = "Hello"
b = "World"
c = a + b
print(c)

age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

```
Hello
e
5
b
a
n
a
n
a
llo
Hello
llo, World!
orl
HELLO, WORLD!
hello, world!
Hello, World!
 Jello, World!
[' Hello', ' World! ']
HelloWorld
My name is John, and I am 36
```

**List**

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

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

#List Length
print(len(thislist))

#The list() Constructor
#It is also possible to use the list() constructor when creating a new list.
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)

#Access Items
print(thislist[1])

#Range of Indexes
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])

#Change Item Value
thislist[1] = "blackcurrant"
print(thislist)

#Append Items
thislist.append("orange")
print(thislist)

#Insert Items
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)

#Extend List

thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)

#Remove Specified Item
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)

#Remove Specified Index
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)

thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)

#Clear the List
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

```
['apple', 'banana', 'cherry']
3
['apple', 'banana', 'cherry']
banana
['cherry', 'orange', 'kiwi']
['apple', 'blackcurrant', 'cherry', 'orange', 'kiwi', 'melon', 'mango']
['apple', 'blackcurrant', 'cherry', 'orange', 'kiwi', 'melon', 'mango', 'orange']
['apple', 'orange', 'banana', 'cherry']
['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
['apple', 'cherry']
['apple', 'cherry']
['banana', 'cherry']
[]
```

**Tuple**

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

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

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

#Tuple Length
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))

#The tuple() Constructor
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)

#Access Tuple Items
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
print(thistuple[-1])

#Range of Indexes
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])

#Add Items
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

```
('apple', 'banana', 'cherry', 'apple', 'cherry')
3
('apple', 'banana', 'cherry')
banana
cherry
('cherry', 'orange', 'kiwi')
```

**Set**

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

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

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

```
thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)
print(len(thisset))
print(type(thisset))

#The set() Constructor
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)

#Access Items
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)

#Add Items
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)

#Remove Item
thisset.remove("banana")
print(thisset)

#Remove a random item by using the pop() method:

thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)

#The clear() method empties the set:

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

#The del keyword will delete the set completely:

thisset = {"apple", "banana", "cherry"}
del thisset
```

```
{'cherry', 'apple', 'banana'}
3
<class 'set'>
{'cherry', 'apple', 'banana'}
cherry
apple
banana
{'orange', 'cherry', 'apple', 'banana'}
{'orange', 'cherry', 'apple'}
cherry
{'apple', 'banana'}
set()
```

**Python If Else Statements – Conditional Statements**

If-Else statements in Python are part of conditional statements, which decide the control of code. As you can notice from the name If-Else, you can notice the code has two ways of directions.

There are situations in real life when we need to make some decisions and based on these decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.

Conditional statements in Python languages decide the direction(Control Flow) of the flow of program execution.

**Types of Control Flow in Python**

Python control flow statements are as follows:

- The if statement
- The if-else statement
- The nested-if statement
- The if-elif-else ladder

```
#If statement:

a = 33
b = 200
if b > a:
  print("b is greater than a")
```

```
b is greater than a
```

### Elif

The elif keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

```
a and b are equal
```

### Else

The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

```
a is greater than b
```

### Nested If

You can have if statements inside if statements, this is called nested if statements.

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

```
 Above ten,
 and also above 20!
```

### Python if-elif-else ladder

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

```
 i is 20
```

### Python if shorthand

```
i = 10
if i < 15: print("i is less than 15")
```

```
i is less than 15
```

Short Hand if-else statement

```
i = 10
print(True) if i < 15 else print(False)
```

```
True
```

**Python Loops**

Python has two primitive loop commands:

- while loops
- for loops

**The while Loop**

With the while loop we can execute a set of statements as long as a condition is true.

```
#Print i as long as i is less than 6:

i = 1
while i < 6:
  print(i)
  i += 1
```

```
1
2
3
4
5
```

**The break Statement**

With the break statement we can stop the loop even if the while condition is true:

```
#Exit the loop when i is 3:

i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

**The continue Statement**

With the continue statement we can stop the current iteration, and continue with the next:

```
#Continue to the next iteration if i is 3:

i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
6
```

**Python while loop with a pass statement**

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

```
a = 'Krishna'
i = 0

while i < len(a):
    i += 1
    pass

print('Value of i :', i)
```

```
Value of i : 7
```

**Python For Loops**

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

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
apple
banana
cherry
```

**Looping Through a String**

```
for x in "banana":
  print(x)
```

```
b
a
n
a
n
a
```

**The break Statement**

```
#Exit the loop when x is "banana":

fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

```
apple
banana
```

**The continue Statement**

With the continue statement we can stop the current iteration of the loop, and continue with the next:

```
#Do not print banana:

fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

```
apple
cherry
```

**Nested Loops**

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```
#Print each adjective for every fruit:

adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

**The pass Statement**

for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```
for x in [0, 1, 2]:
  pass
```

**Python Functions**

A function is a block of code which only runs when it is called.

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

A function can return data as a result.

**Some Benefits of Using Functions**

- Increase Code Readability
- Increase Code Reusability

**Types of Functions in Python**

There are mainly two types of functions in Python.

1. Built-in library function: These are Standard functions in Python that are available to use.
2. User-defined function: We can create our own functions based on our requirements.

```
def my_function():
  print("Hello from a function")

my_function()
```

```
Hello from a function
```

**Python Function with Parameters**

```
def add(num1: int, num2: int) -> int:
    """Add two numbers"""
    num3 = num1 + num2

    return num3

# Driver code
num1, num2 = 5, 15
ans = add(num1, num2)
print(f"The addition of {num1} and {num2} results {ans}.")
```

```
The addition of 5 and 15 results 20.
```

**Python Function Arguments**

```
# A simple Python function to check
# whether x is even or odd
def evenOdd(x):
    if (x % 2 == 0):
        print("even")
    else:
        print("odd")


# Driver code to call the function
evenOdd(2)
evenOdd(3)
```

```
even
odd
```

**Types of Python Function Arguments** Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

1. Default argument
2. Keyword arguments (named arguments)
3. Positional arguments
4. Arbitrary arguments (variable-length arguments *args and* *kwargs)

**Default Arguments**

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

```
# default arguments
def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)


# Driver code (We call myFun() with only argument)
myFun(10)
```

```
x:  10
y:  50
```

**Keyword Arguments**

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

```
def student(firstname, lastname):
    print(firstname, lastname)


# Keyword arguments
student(firstname='krishna', lastname='Patle')
```

```
krishna Patle
```

**Positional Arguments**

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)

nameAge("Suraj", 27)
```

```
Hi, I am Suraj
My age is  27
```

**Arbitrary Keyword Arguments**

In Python Arbitrary Keyword Arguments, *args, and *kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args in Python (Non-Keyword Arguments)
- **kwargs in Python (Keyword Arguments)

```
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print(arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

```
Hello
Welcome
to
GeeksforGeeks
```

```
# *kwargs for variable number of keyword arguments


def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))


# Driver code
myFun(first='Geeks', mid='for', last='Geeks')
```

```
first == Geeks
mid == for
last == Geeks
```

**Anonymous Functions in Python**

In Python, an anonymous function means that a function is without a name. As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

```
# Python code to illustrate the cube of a number
# using lambda function
def cube(x): return x*x*x

cube_v2 = lambda x : x*x*x

print(cube(7))
print(cube_v2(7))
```

```
343
343
```

**Recursive Functions in Python**

Recursion in Python refers to when a function calls itself. There are many instances when you have to build a recursive function to solve Mathematical and Recursive Problems.

Using a recursive function should be done with caution, as a recursive function can become like a non-terminating loop. It is better to check your exit statement while creating a recursive function.

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

print(factorial(4))
```

```
24
```

**Python OOPs Concepts**

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the

data and the functions that work on that together as a single unit so that no other part of the code can access this data.

**OOPs Concepts in Python**

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

**Python Classes/Objects**

Python is an object oriented programming language.

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

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

To create a class, use the keyword class():

```
class MyClass:
  x = 5

print(MyClass)
```

```
<class '__main__.MyClass'>
```

**Create Object**

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

```
#Create an object named p1, and print the value of x:

p1 = MyClass()
print(p1.x)
```

```
5
```

**The Python **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. Now let us define a class and create some objects using the self and **init** method.

```
#Create a class named Person, use the __init__() function to assign values for name and age:

class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

**Python Inheritance**

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Usage
```

Inheritance allows a new class (Dog and Cat) to inherit attributes and methods from an existing class (Animal). This promotes code reusability and establishes a relationship between classes.

```
print(dog.speak())  # Output: Woof!
```

**Polymorphism:**

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def animal_speak(animal):
    return animal.speak()

# Usage
dog = Dog()
cat = Cat()

print(animal_speak(dog))  # Output: Woof!
print(animal_speak(cat))  # Output: Meow!
```

```
Woof!
Meow!
```

Polymorphism allows objects of different classes to be treated as objects of a common base class. In the example, both Dog and Cat classes have a speak method. The animal_speak function takes any object with a speak method, demonstrating polymorphism.

**Encapsulation**

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the