Introduction

Introduction to Python By {Coding Ninjas}

Python is an easy-to-learn and powerful Object-Oriented Programming language. It is a high-level programming language.

Why Python?

1. Easy to Use: Python is comparatively an easier-to-use language as compared to other programming languages.
2. Expressive Language: The syntax of Python is closer to how you would write pseudocode. Which makes it capable of expressing the code’s purpose better than many other languages.
3. Interpreted Language: Python is an interpreted language; this means that the Python installation interprets and executes the code a line at a time.
4. Python is one of the most popular programming languages to be used in Web Development owing to the variety of Web Development platforms built over it like Django, Flask, etc.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has a syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Good to know

- The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- In this tutorial, Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans, or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

- Python was designed for readability and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions, and classes. Other programming languages often use curly brackets for this purpose.

Python Download

Many PCs and Macs will have python already installed.
To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python --version
```

To check if you have python installed on a Linux or Mac, then on Linux open the command line or on Mac open the Terminal, and type:

```
python --version
```

In case you have not found:-
The very first step towards Python Programming would be to download the tools required to run the Python language. We will be using Python 3 for the course. You can download the latest version of Python 3 from https://www.python.org/downloads/

Note:- If you are using Windows OS, then while installing Python make sure that "Add Python to PATH" is checked.

Getting an IDE for writing programs:
You can use any IDE of your choice, however, you are recommended to use Jupyter Notebook. You can download it from https://jupyter.org/install.

Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.
The way to run a python file is like this on the command line:

```
C:\Users\Your Name>python helloworld.py
```

Where "helloworld.py" is the name of your python file.
Let's write our first Python file, called helloworld.py, which can be done in any text editor.

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

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

```
C:\Users\Your Name>python helloworld.py
```

The output should read:

```
Hello, World!
```

The Python Command Line

To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command-line itself.

Type the following on the Windows, Mac, or Linux command line:

```
C:\Users\Your Name>python
```

Or, if the "python" command did not work, you can try "py":

```
C:\Users\Your Name>py
```

From there you can write any python, including our hello world example from earlier in the tutorial:

```
C:\Users\Your Name>python

Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
```

Which will write "Hello, World!" in the command line:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

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

Previous

Next

Python Comments

Python Comments

1. Comments can be used to explain Python code.
2. Comments can be used to make the code more readable.
3. Comments can be used to prevent execution when testing code.

Creating a Comment

Comments start with a '#', and Python will ignore them:

```python
# This is a how we write a comment in python.
print("Hello, World!")
```

Multi-Line Comments

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

```python
# This way we can
# Write comments in
# Multiple lines.
print("Hello, World!")
```

Or, not quite as intended, you can use a multiline string.

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

```python
"""
Another way to write
comments in more than one
Line.
"""
print("Hello, World!")
```

Note: We can achieve multi-line comments by using single triple quotes or by double triple quotes.

Previous

Next

Variables

Introduction to Variables

A **variable** in Python represents a named location that refers to a value. These values can be used and processed during the program run. In other words, variables are labels/names to which we can assign values and use them as a reference to those values throughout the code.

Variables are fundamental to programming for two reasons:

● **Variables keep values accessible:** For example, the result of a time-consuming operation can be assigned to a variable so that the same operation is not performed every time when we look for the result.
● **Variables give values context:** For example, the number 56 could mean a lot of different things, such as the number of students in a class, or the average weight of all students in the class. Assigning the number 56 to a variable with a name like num_students would make more sense, to distinguish it from another variable average_weight, which would refer to the average weight of the students. This way we can have different variables pointing to different values.

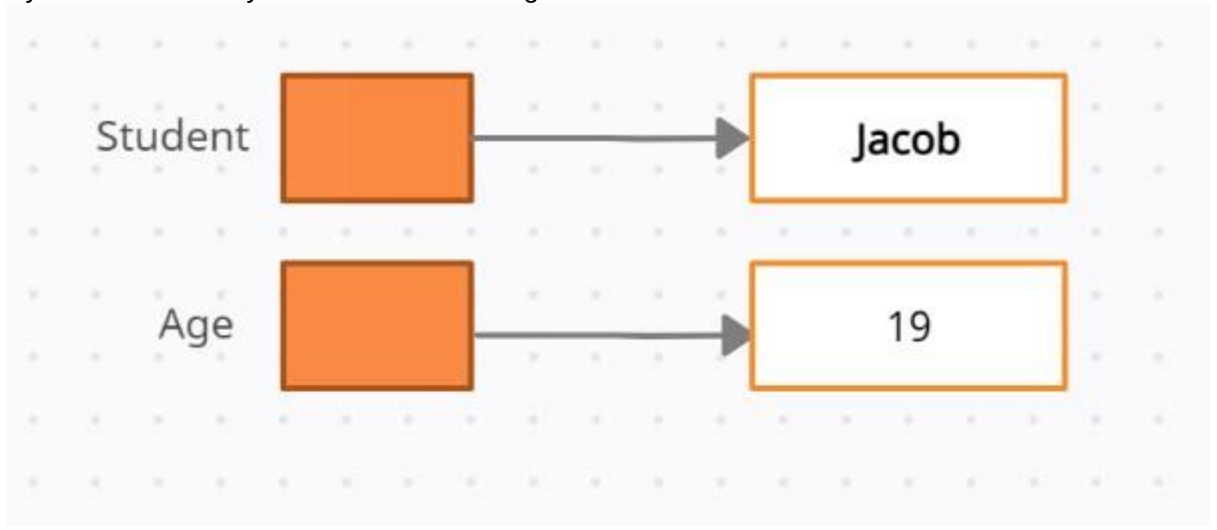
How are Values Assigned to A Variable?

Values are assigned to a variable using a special symbol "=", called the assignment operator. An operator is a symbol, like = or +, that performs some operation on one or more values. For example, the + operator takes two numbers, one to the left of the
operator and one to the right, and adds them together. Likewise, the "=" operator takes a value to the right of the operator and assigns it to the name/label/variable on the left of the operator.

For Example: Now let us create a variable namely **Student** to hold a student's name and a variable **Age** to hold a student's age.

```
>>> Student = "Jacob"
>>> Age = 19
```

Python will internally create labels referring to these values as shown below:



Now, let us modify the first program we wrote.

```python
greeting = "Hello, World!"
print(greeting)
```

Here, the Python program assigned the value of the string to a variable greeting, and then when we call print(greeting), it prints the value that the variable, greeting, points to i.e. "Hello, World!"

We get the output as:-

```
'Hello, World!'
```

Naming a Variable

You must keep the following points in your mind while naming a variable:-

- Variable names can contain letters, numbers, and underscores.
- They cannot contain spaces.
- Variable names cannot start with a number.
- Variable names are case-sensitive. For example:- The variable names Temp and temp are different.
- While writing a program, creating self-explanatory variable names helps a lot in increasing the readability of the code. However, too long names can clutter up the program and make it difficult to read.

Previous

Next

DataTypes

Introduction to data types

Data types are the classification or categorization of data items. Data types represent a kind of value that determines what operations can be performed on that data. Numeric, non-numeric, and Boolean (true/false) data are the most used data types. However, each programming language has its classification largely reflecting its programming philosophy. Python offers the following built-in data types:

- Numbers
    - Integers
    - Floating Point Numbers
    - Complex Numbers
- Strings
- Boolean Values
- List, Tuple, and Dictionary

Note:- If a variable has been assigned a value of some data type, It can be reassigned as a value belonging to some other Data Type in the future.

```
a= "Raw" # String Data Type
a= 10 # Integer Data Type
a= 5.6 # Floating Point Number Data Type
a= 1+8j # Complex Number
a= True # Boolean Value
```

Introduction to Python Numbers

Number data types store numerical values. Python supports Integers(defined as int), floating-point numbers(defined as float), and complex numbers(complex classes).

● Integers can be of any length (Only limited by the memory available). They do have a decimal point and can be positive or negative.
● A floating-point number is a number having a fractional part. The presence of a decimal point indicates a floating-point number. They have a precision of up to 15 digits.
● 1 is an integer, 1.0 is a floating-point number.
● Complex numbers have an imaginary part and are of the form, x + yj (where x is the real part and y is the imaginary part).

To know which class a variable belongs to we can use the type() function. Similarly, you can use the instance() function to check if an object belongs to a particular class.
Here are a few examples:-

```
b = 5
print(b, "is of type", type(b))
b = 2.0
print(b, "is of type", type(b))
b = 1+2j
print(b, "is complex number?", isinstance(b,complex))
```

And we will get the output as:

```
5 is of type <class ‘int’>

2.0 is of type <class ‘float’>

1+2j is a complex number? True
```

Previous

Next

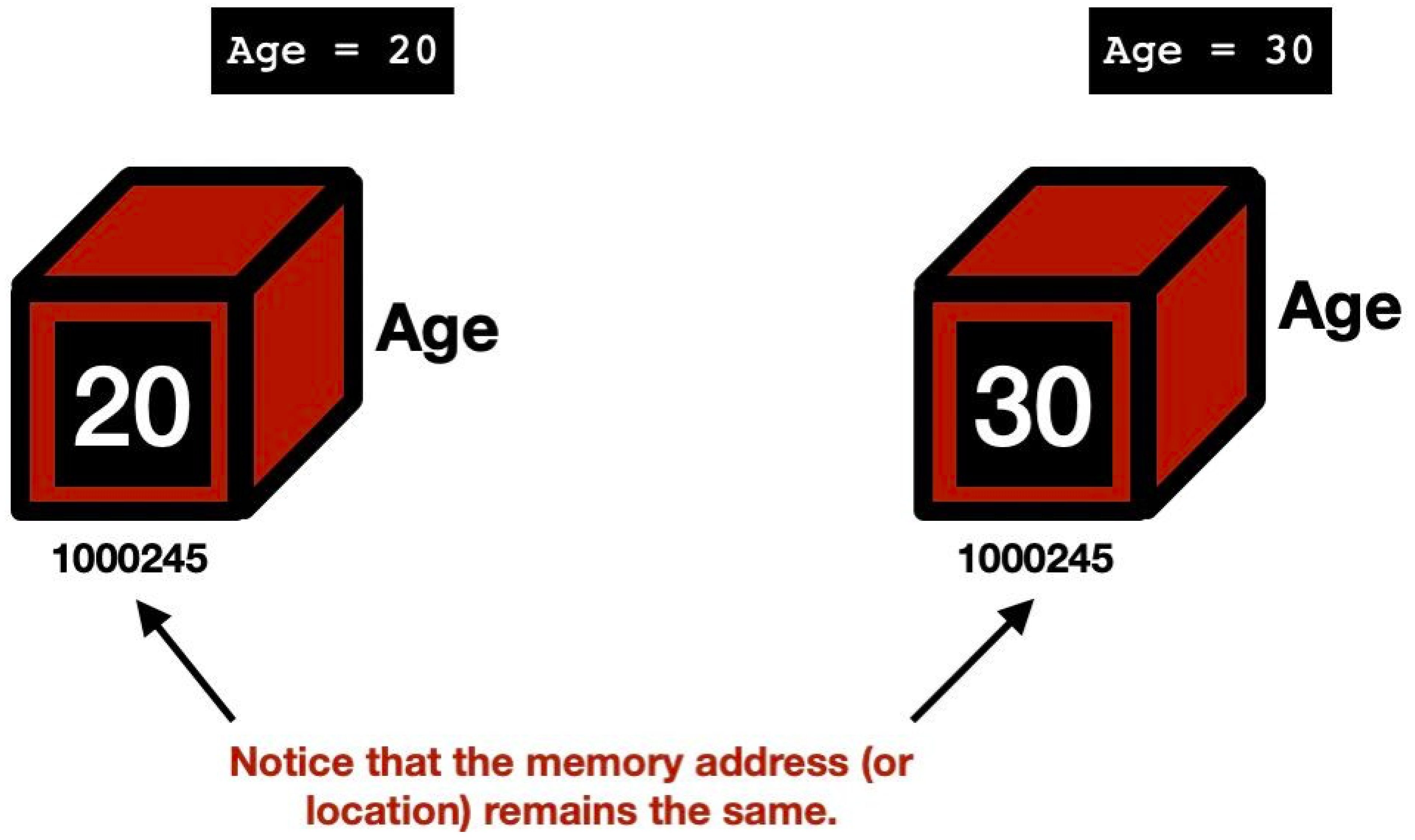
Variables in Memory

Traditional Programming Languages’ Variables in Memory

Let us study how variables and the values they are assigned, are represented in memory, in traditional programming languages like C, C++, Java, etc. In these languages, variables are like storage containers. They are like named storage locations that store some value. In such cases, whenever we declare a new variable, a new storage location is given to that name/label and the value is stored at that named location. Now, whenever a new value is reassigned to that variable, the storage location remains the same. However, the value stored in the storage location is updated. This can be shown from the following illustration.

Consider the following script:

```
Age = 20
Age = 30 # Re-assigning a different value to the same variable
```

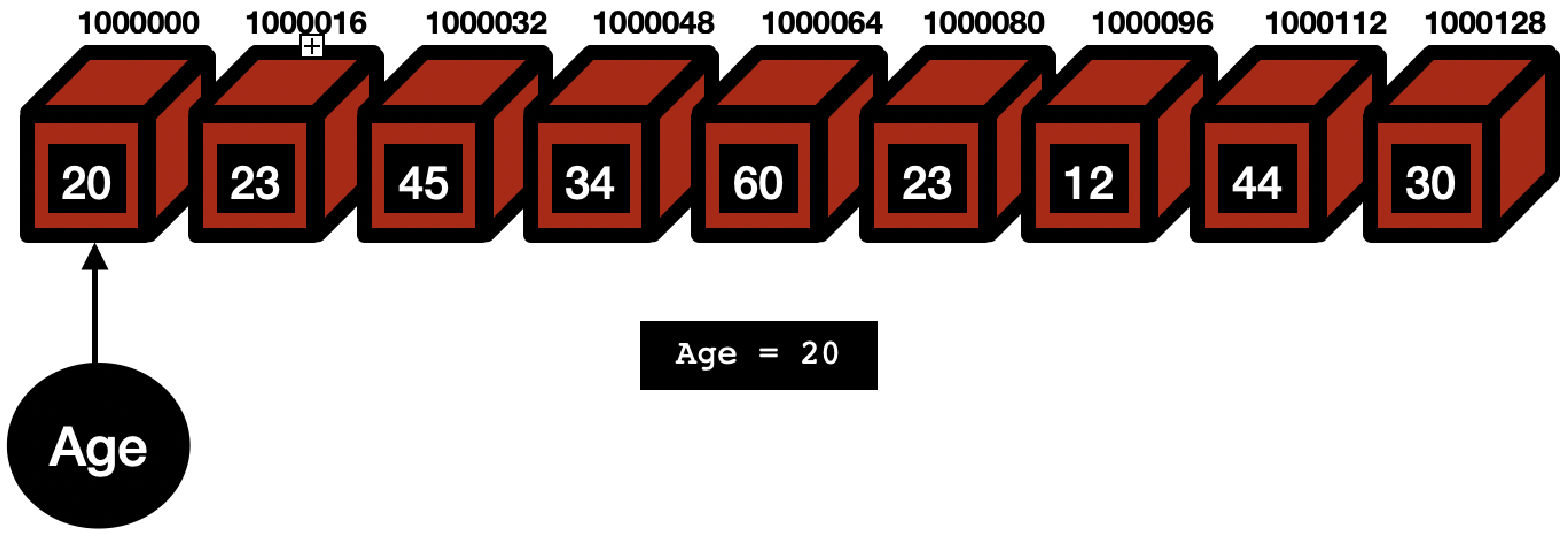


Age = 20
Age = 30
20
Age
1000245
30
Age
1000245
Notice that the memory address (or location) remains the same.

In the above script, when we declare a new variable Age, a container box/ Memory Location is named Age and the value 20 is stored in the memory address 1000245 with the name/label, Age. Now, on reassigning the value 30 to Age, the value 30 is stored in the same memory location. This is how the variables behave in Traditional programming languages.

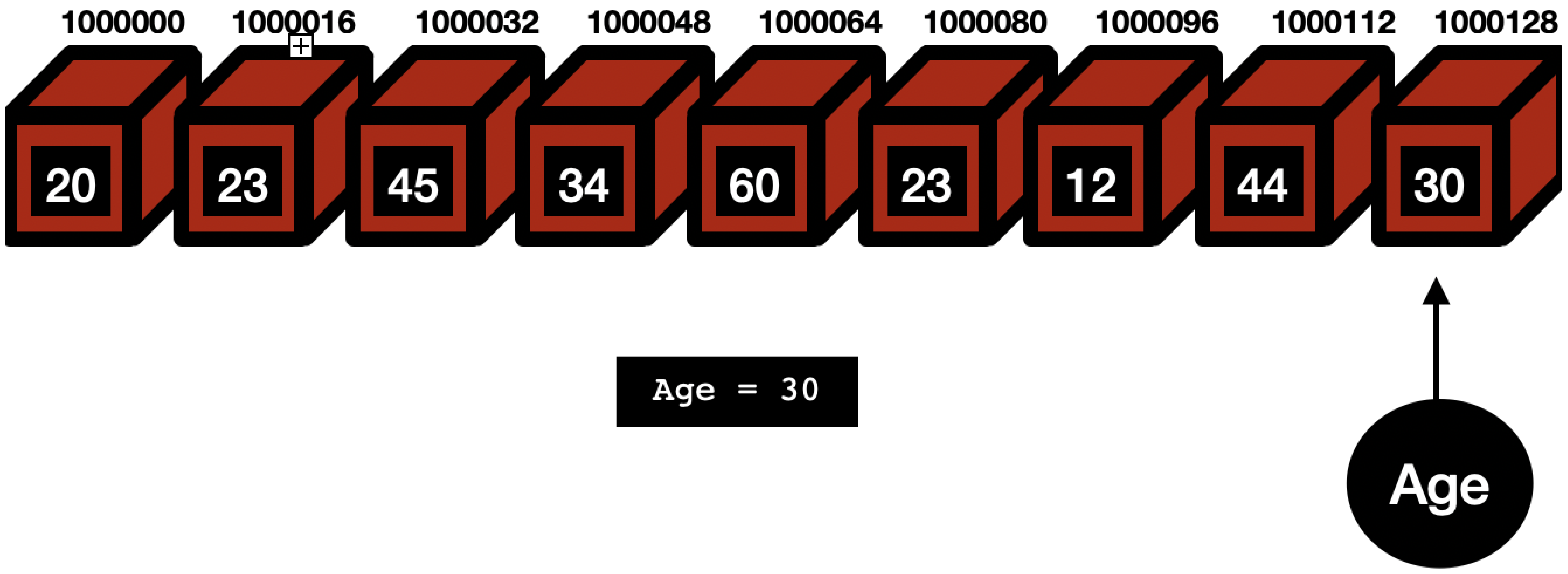
Python Variables in Memory

Python variables are not created in the form most other programming languages do. These variables do not have fixed locations, unlike other languages. The locations they refer/point to changes every time their value changes.
Python preloads some commonly used values in an area of memory. This memory space has values/literals at defined memory locations and all these locations have different addresses.
When we give the command, Age = 20, the variable Age is created as a label pointing to a memory location where 20 is already stored. If 20 is not present in any of the memory locations, then 20 is stored in an empty memory location with a unique address and then the Age is made to point to that memory location.



Now, when we give the second command, Age = 30, the label Age will not have the same location as earlier. Now it will point to a memory location where 30 is stored. So this time the memory location for the label Age is changed.

.



One interesting thing to note while working with Python variables is that when we create a variable, we actually create an object somewhere in the memory with a unique mapping or ID/Address. We can see this unique ID generated against each object using ***id().***

Example:

```
a = 5          #An object, "a" is created that has a unique ID/memory block which stores 5 as a variable
b = 10         #An object, "b" is created that has a unique ID/memory block which stores 10 as a variable
print(id(a))   #Printing the unique ID for "a"
print(id(b))   #Printing the unique ID for "b"
```

output:

```
4425656800
4425656960
```

Another interesting thing to note is, when we create multiple objects with the same value and type, instead of creating new memory blocks for each of them, they all point to the very first version of the object holding the same value and type.

Example:

```
a = 10
b = 10
print(id(a))
print(id(b))
```

output:

```
4425656960
4425656960
```

Previous

Next

Variables Scope

Scope of Variables

All variables in a program may not be accessible at all locations in that program. Part(s) of the program within which the variable name is legal and accessible, is called the scope of the variable. A variable will only be visible to and accessible by the code blocks in its scope.
There are broadly two kinds of scopes in Python −

- Global scope
- Local scope

Global Scope :

A variable/name declared in the top-level segment **(__main__)** of a program is said to have a global scope and is usable inside the whole program (Can be accessed from anywhere in the program).
In Python, a variable declared outside a function is known as a global variable. This means that a global variable can be accessed from inside or outside of the function.

Creating a Global Variable

Consider the given code snippet:

```
x = "Global Variable"
def foo():
    print("Value of x: ", x)
foo()
```

Here, we created a global variable **x = "Global Variable"**. Then, we created a function foo to print the value of the global variable from inside the function. We get the output :

```
Global Variable
```

Thus we can conclude that we can access a global variable from inside any function.
What if you want to change the value of a Global Variable from inside a function?

Consider the code snippet:

```
x = "Global Variable"
def foo():
    print("Value of x: ", x)
foo()
```

In this code block, we tried to update the value of the global variable x. We get an output as:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

This happens because, when the command **x=x-1**, is interpreted, Python treats this **x** as a local variable and we have not defined any local variable **x** inside the function foo().

Previous

Next

LocalVariable

Local Scope:

The variables which are defined inside a function body have a local scope. This implies that local variables can be accessed only inside the function in which they are declared.

Creating a Local Variable

We declare a local variable inside a function. Consider the given function definition:

```
def func():
    name = "Shivam Kumar Singh"
        print(name)

func()
```

We get the output as:

```
Shivam Kumar Singh
```

Accessing A Local Variable Outside The Scope

```
def func():
    name = "Shivam Kumar Singh"

func()
print(name)
```

In the above code, we declared a local variable y inside the function foo(), and then we tried to access it from outside the function. We get the output as:

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

We get an error because the lifetime of a **local variable** is the function it is defined in.

Outside the function, the variable does not exist and cannot be accessed. In other words, a variable cannot be accessed outside its scope.

The Lifetime of a Variable

The time for which the variable exists in memory is called the lifetime of that variable:

- The lifetime of a Global variable is the entire program run (i.e., they live in the memory as long as the program is being executed).
- The lifetime of a Local variable is their function's run (i.e., as long as their function is being executed).

Previous

Next

GlobalVariable

Global Variable And Local Variable With The Same Name

Consider the code given:

```
val = 1
def func():
  val = 50
  print("Value of Local variable:", val)

func()
```

```
print("Value of Global Variable:", val)
```

In this, we have declared a global variable val **= 1** outside the function func(). Now, we re-declared a local variable inside the function func**()** with the same name val. Now, we try to print the values of val, inside, and outside the function. We observe the following output:

```
Value of Local variable:: 50
Value of Global variable:: 1
```

In the above code, both global and local variables have the same name val. On printing the value of val we get a different result, because the variables have been declared in different scopes, i.e., the local scope inside func() and global scope outside func().

When we print the value of the variable inside **func**() it outputs **Value of Local variable: 10**. This is the local scope of the variable. In the local scope, it prints the value that it has been assigned inside the function.

Similarly, when we print the variable outside **func**(), it outputs **Value of Global variable: 5**. This is the global scope of the variable, and the value of the global variable val is printed.

Previous

Next

TypeCasting

Typecasting

There may be times when you want to specify a type on to a variable. This can be done with type casting. Python is an OOP language, and as such, it uses classes to define data types, including its primitive types.

Type casting is the method to convert from data type to another data type, according to user's requirements. This article introduces to the different type of casting supported by Python and how to implement them.

Python supports two types of type casting –

- Implicit Type Conversion
- Explicit Type Conversion

1. Implicit Type Conversion:

In this, method, Python automatically converts one data type into another data type. There is no user intervention.

```
val = 5
print(type(val))
```

output:

```
#Python automatically converts val to int.
< class 'int' >
```

Example:

```
a = 8.0
b = 4.0
m = a * b
print(m)
print(type(m))
```

output:

```
32.0
< class 'float'
```

2. Explicit Type Conversion

In Explicit Type Conversion user needs to manually convert data types.
Mainly in typecasting can be done with these data type function:

- **int()** - int class to construct an integer literal.
- **float()** - float class to construct a float literal.
- **str()** - string class to construct a string literal.

Let's see some example of typecasting:

Example:

```
# int variable
a = 5.9

# typecast to int
n = int(a)
print(n)
print(type(a))
```

output:

```
5
<class 'int'>
```

Here, we are casting float into int data type with int**()** function.

Previous

Next

Print Function

Basic I/O in Python

Python print() Function

The print() function prints the specified message to the screen or other standard output device.

The message can be a string or any other object.The object will be converted into a string before written to the screen.

The simplest way to produce output is using the print() function where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string before writing to the screen.

Syntax:

```
print(object(s), sep=separator, end=end, file=file, flush=flush)
```

Example:

Printing a message on the screen:
```
print("Hello  World")
```

Output:

```
Hello World
```

Example:

```
print ("Codingninjas is best for DSA Content.")
```

output:

```
Codingninjas is best for DSA Content.
```

Example:

Printing more than one object:
```
print("Hello", "how are you?")
```

Output:

```
Hello how are you?
```

Example:

```
a = 5
b = 7
print(a,b)
```

output:

```
5 7
```

Python end parameter in print()

By default python’s print() function ends with a new line. A programmer with C/C++ background may wonder how to print without a new line.

Python’s print() function comes with a parameter called ‘end’. By default, the value of this parameter is ‘\n’, i.e. the new line character. You can end a print statement with any character/string using this parameter.

Example:

```
# This Python program must be run with
# Python 3 as it won't work with 2.7.
# ends the output with a <space>

print("Welcome to" , end = ' ')
print("CodingNinjas", end = ' ')
```

output:

```
Welcome to CodingNinjas
```

Example:

```
# This Python program must be run with
# Python 3 as it won't work with 2.7.
# ends the output with a <space>

print("Welcome" , end = '@')
print("CodingNinjas", end = ' ')
```

output:

```
Welcome@CodingNinjas
```

Python sep parameter in print()

The separator between the arguments to print() function in Python is space by default (soft space feature), which can be modified and can be made to any character, integer, or string as per our choice. The ‘sep’ parameter is used to achieve the same, it is found only in python 3.x or later. It is also used for formatting the output strings.

Example:

```
# Code for disabling the softspace feature
print('CodingNinjas', sep='')

# For formatting a date
print('01','01','2021', sep='-')

# Another example
print('saif','codingninjas', sep='@')
```

output:

```
CodingNinjas
01-01-2021
saif@codingninjas
```

Previous

Next

Input

input()

To take input from the user, we use the **input()** function.

input(): This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether the user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python.

Example:

```
# Python program showing the use of input()

val = input("Enter your name: ")
print(val)
```

Output:

```
Enter your name: saif
'saif'
```

How the input function works in Python :

- When the **input()** function executes, program flow will be stopped until the user has given input.
- The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
- Whatever you enter as input, the input function converts it into a string. if you enter an integer value still **input()** function convert it into a string. You need to explicitly convert it into an integer in your code using **typecasting.**

Example:

```
# Program to check input type in Python

name = input ("Enter name :")
print(name)
print ("type of name", type(name))
```

Output:

```
Enter your name: saif
'saif'
type of name <class 'str'>
```

Example:

```
# Program to check input type in Python
```

```
num = int(input ("Enter number :"))
print(num)
print ("type of num", type(num))
```

Output:

```
Enter number : 10
10
type of num <class 'int'>
```

How to take space-separated input in one line in Python?

```
x, y = input().split()
```

Note that we don't have to explicitly specify split(' ') because split() uses any whitespace characters as a delimiter as default. One thing to note in the above Python code is, both x and y would be of string. We can convert them to int using another line

Example:

```
# Program take space-separated input in one line in Python

x, y = input().split()
print(x, y)
```

output:

```
10 20
10 20
```

Example:

```
# Program to check input type in Python

x, y = input().split()
a = input()
print ("type of x", type(x))
print ("type of y", type(y))
print ("type of a", type(a))
```

output:

```
10 20
Parikh
type of x <class 'str'>
type of y <class 'str'>
type of a <class 'str'>
```

Question: Add two numbers.

```
print("Enter the value of a")
a = int(input())
print("Enter the value of b")
b = int(input())
c = a + b
print("value of c is :", c)
```

output:

```
Enter the value of a
2
Enter the value of b
3
value of c is : 5
```

Example:

```
n = input()
print(n)
Input: Hello world
Output: 'Hello World'
```

Previous
Next
Introduction

Introduction to Operators

Operators are used to performing operations on variables and values.

In the example below, we use the + operator to add together two values:

Example:

```
print(10 + 5)
```

Python divides the operators into the following groups:

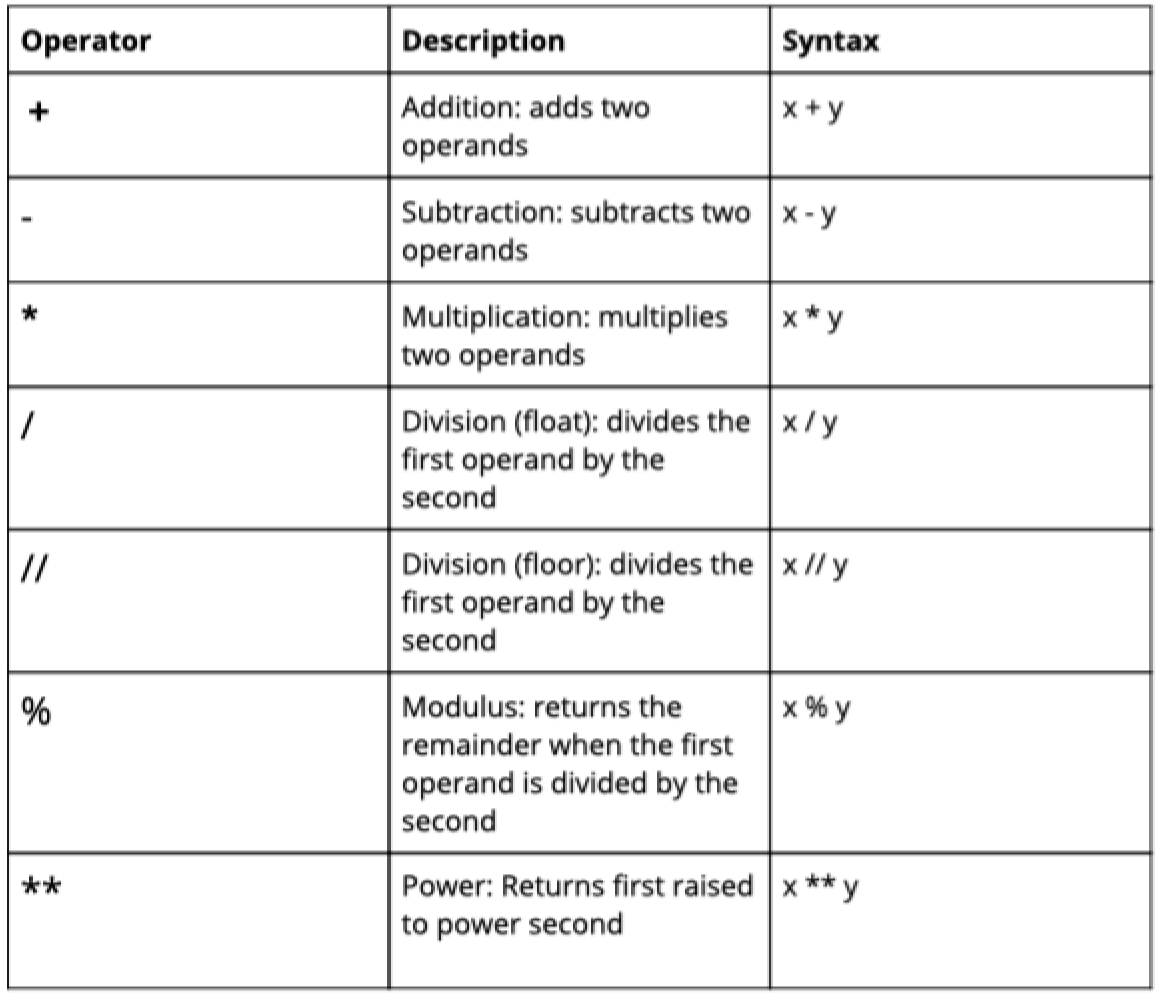
- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Previous
Next
Notes

Python Arithmetic Operators

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

1. Arithmetic operators:



Operator	Description	Syntax
+	Addition: adds two operands	x + y
-	Subtraction: subtracts two operands	x - y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Arithmetic operators are used to performing mathematical operations like addition, subtraction, multiplication, and division.

Example:

Addition:

```
#Addition of a number

a = 10
b = 5
add = a + b
print(add)
```

output:

```
15
```

Subtraction:

```
#Subtraction of a number

a = 10
b = 5
sub = a - b
print(sub)
```

output:

```
5
```

Multiplication:

```
#Multiplication of a number

a = 10
b = 5
mul = a * b
print(add)
```

output:

```
50
```

Division(floor):

```
#Division(floor) of a number
a = 13
b = 5
div = a // b
print(div)
```

output:

```
2
```

Division(float):

```
#Division(float) of a number
a = 10
b = 5
div = a / b
print(div)
```

output:

```
2.0
```

Modulo(remainder):

```
#Modulo of both number

a = 10
b = 5
mod = a % b
print(mod)
```

output:

```
0
```

Power:

```
#Modulo of both number

a = 2
b = 3
pow = a ** b
print(pow)
```

output:

```
8
```

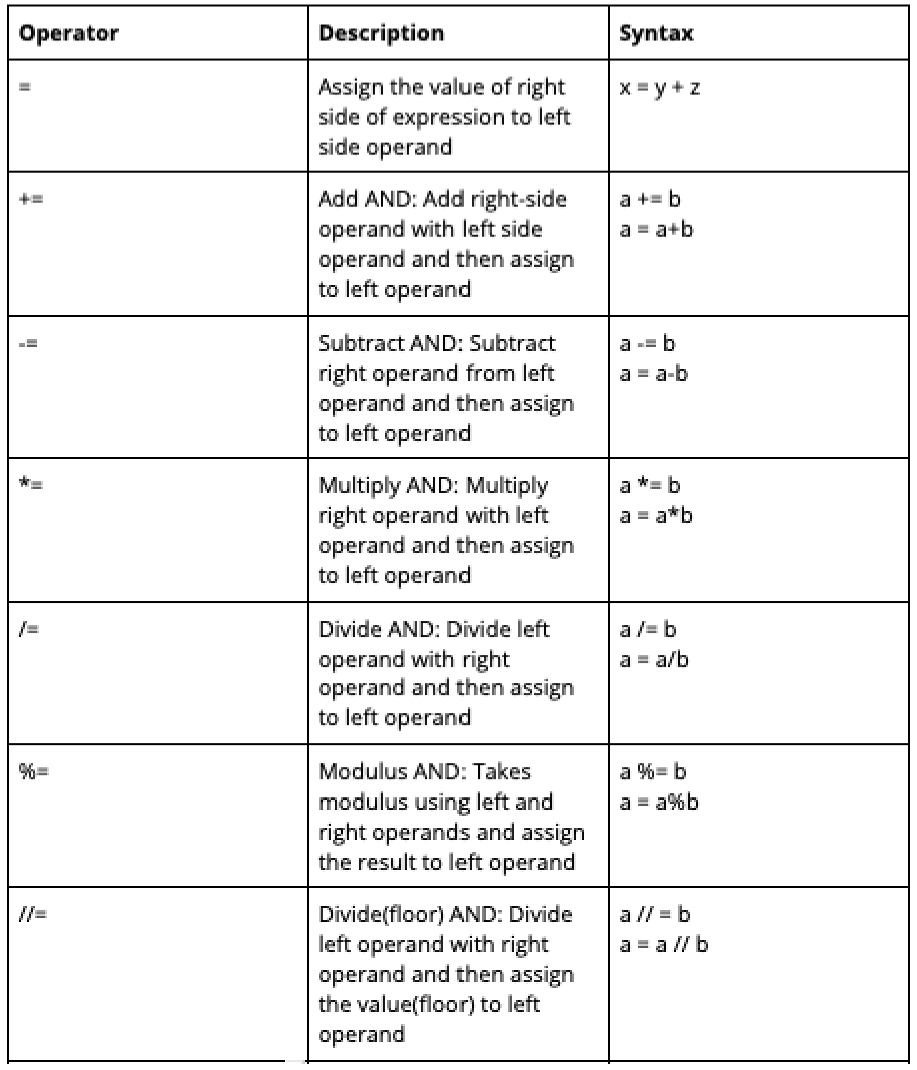
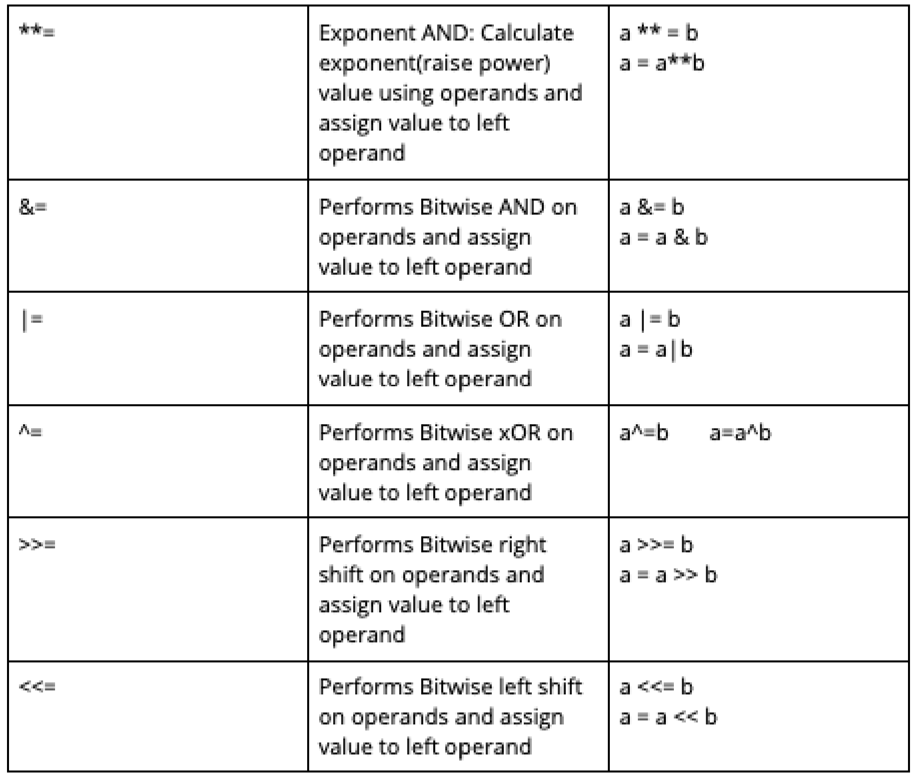
Previous

Next

Notes

Python Assignment Operators

Assignment operators are used to assigning values to variables:

Operator	Description	Syntax
=	Assign the value of right side of expression to left side operand	x = y + z

+=	Add AND: Add right-side operand with left side operand and then assign to left operand	a += b
		a = a+b

-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a -= b
		a = a-b

*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a *= b a = a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a /= b a = a/b
%=	Modulus AND: Takes modulus using left and right operands and assign the result to left operand	a %= b a = a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a // = b a = a // b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a ** = b a = a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a &= b a = a & b
|=	Performs Bitwise OR on operands and assign value to left operand	a |= b a = a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a >>= b a = a >> b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a = a << b

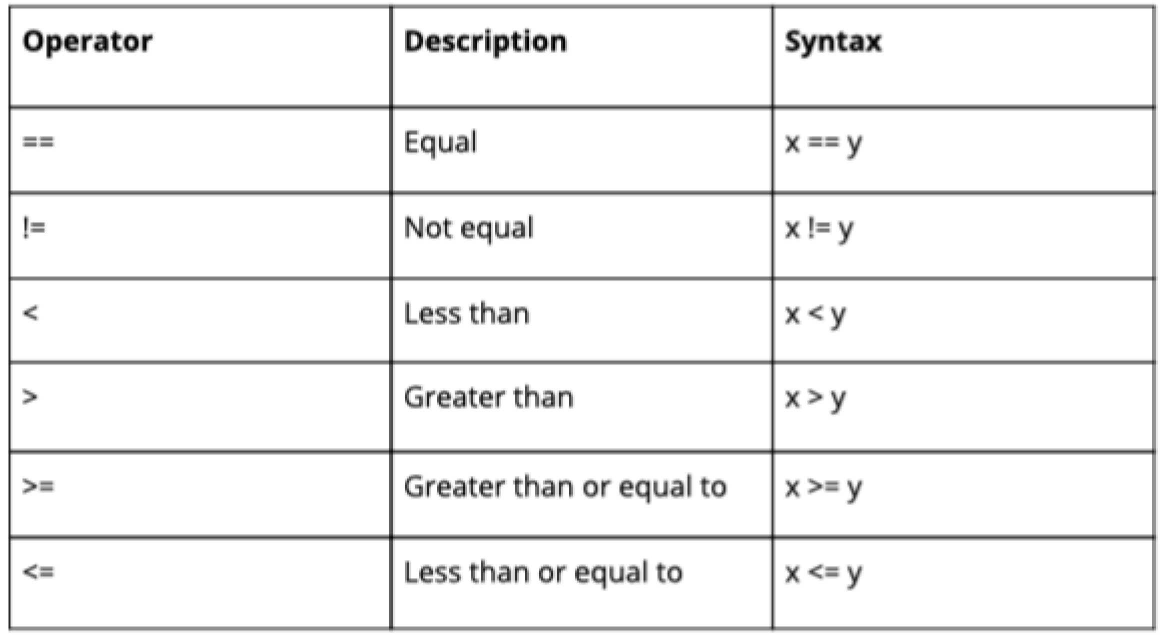
Previous

Next

Notes

Python Comparison Operators

Comparison operators compare the values. It either returns **True** or **False** according to the condition



Operator	Description	Syntax
==	Equal	x == y
!=	Not equal	x != y
<	Less than	x < y
>	Greater than	x > y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Example:

```
# Examples of Comparison Operators

a = 23
b = 11

# a > b is True
print(a > b)

# a < b is False
print(a < b)

# a == b is False
print(a == b)

# a != b is True
print(a != b)

# a >= b is True
print(a >= b)
```

```
# a <= b is False
print(a <= b)
```

output:

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

Previous

Next

Notes

Python Logical Operators

Logical operators perform Logical **AND**, Logical **OR**, and Logical **NOT** operations.

Operator	Description	Syntax
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if an operand is false	not x

The truth table for all combinations of values of X and Y.

X	Y	X and Y	X or Y	not(X)	not(Y)
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Example:

```
# Examples of Logical Operator
x = True
y = False

# Print x and y is False
print(x and y)

# Print x or y is True
print(x or y)

# Print not x is False
print(not x)
```

output:

```
False
```

```
True
False
```

Previous

Next

Notes

Python Identity Operators

is and **is not** are the identity operators. It compares the reference id of the objects.

```
is          Returns true if the operands are referring to the same object.
is not      Returns true if the operands are not referring to the same object.
```

Example:

```
# Examples of Identity operators
a = 10
b = 10

name1 = 'CodeStudio'
name2 = 'CodeStudio'

list1 = [11,22,33]
list2 = [11,22,33]

print(a is b)
print(name1 is not name2)

# Output is False since lists are mutable.
print(list1 is list2)
```

output:

```
True
False
False
```

Python Membership Operators

The membership operators in Python are **in** and **not in** which are used to test if a value exists in a sequence. The uses are :

```
in            returns True only if the value is found in the sequence
not in        returns True only if the value is not found in the sequence
```

Example:

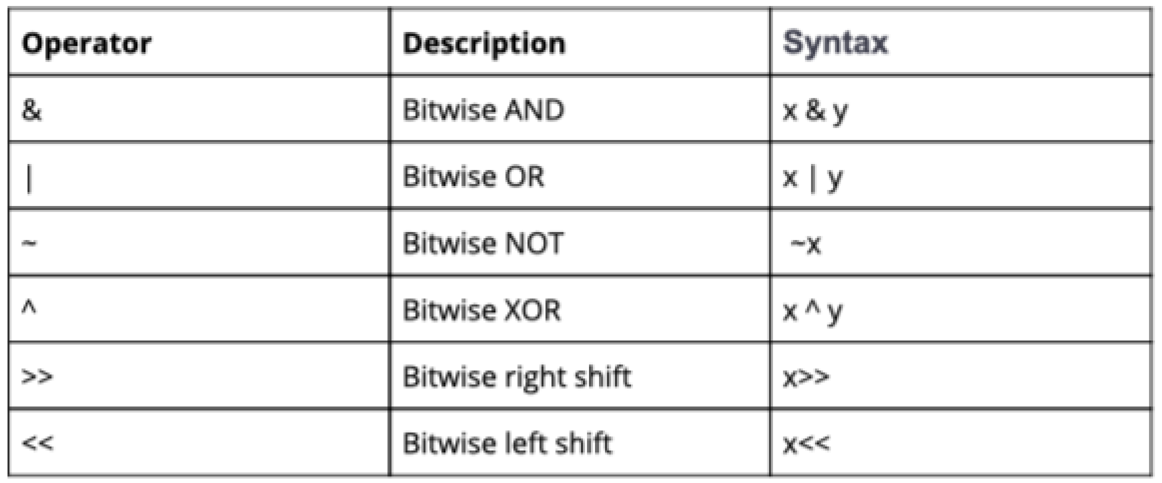
```
# Examples of Membership operator

x = 'CodeStudio'
list = [1,2,3]

print('S' in x)
print('codezen' not in x)
print(3 in list)
print('d' in x)
```

output:

```
True
True
True
True
```

Python Bitwise Operators

Bitwise operators act on bits and perform bit by bit operations.



Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Example:

```
# Examples of Bitwise operators
a = 20
b = 8

# Print bitwise AND operation
print(a & b)

# Print bitwise OR operation
print(a | b)

# Print bitwise NOT operation
print(~a)

# print bitwise XOR operation
print(a ^ b)

# print bitwise right shift operation
print(a >> 2)

# print bitwise left shift operation
print(a << 2)
```

output:

```
0
28
-21
```

```
28
5
80
```

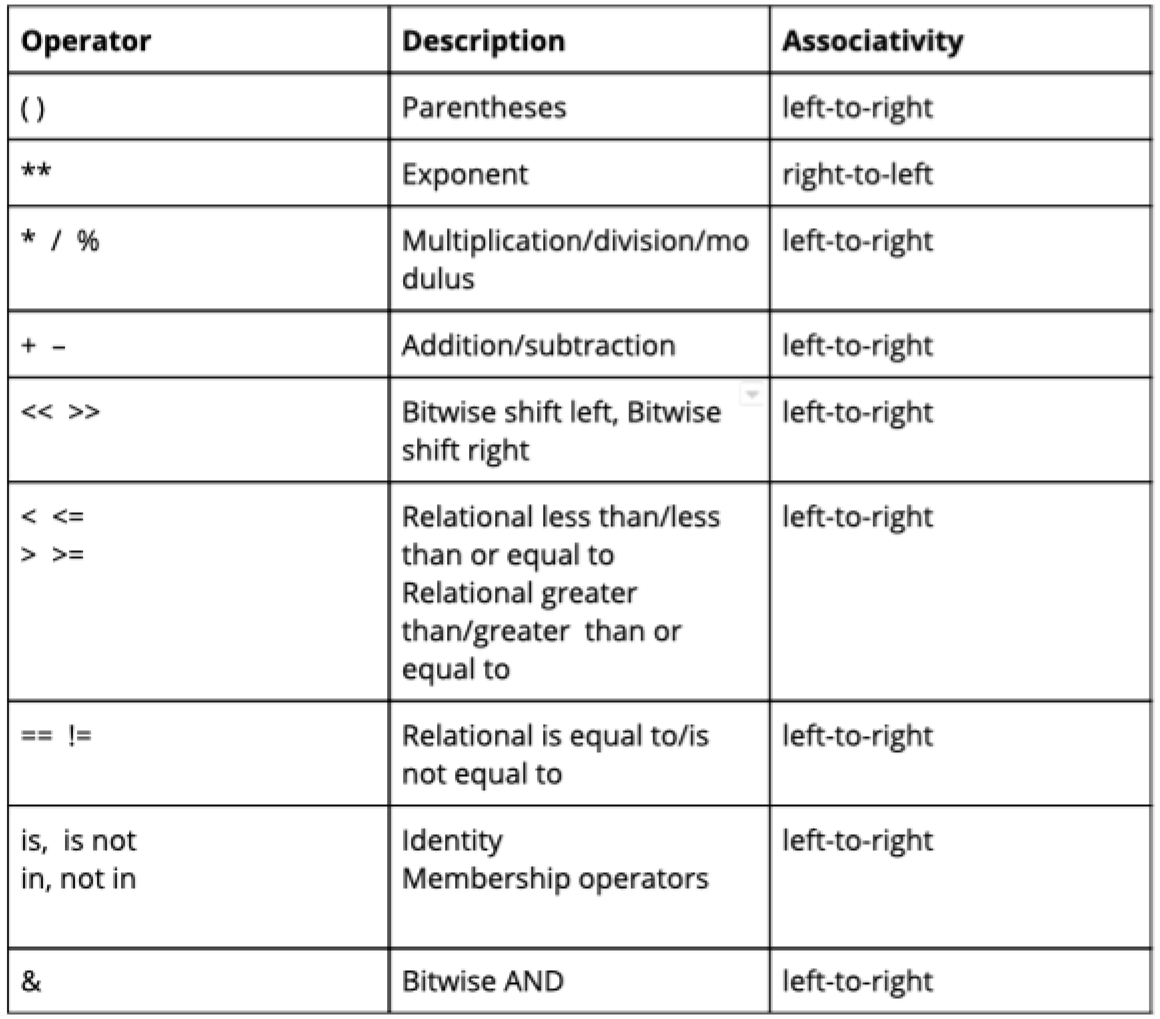
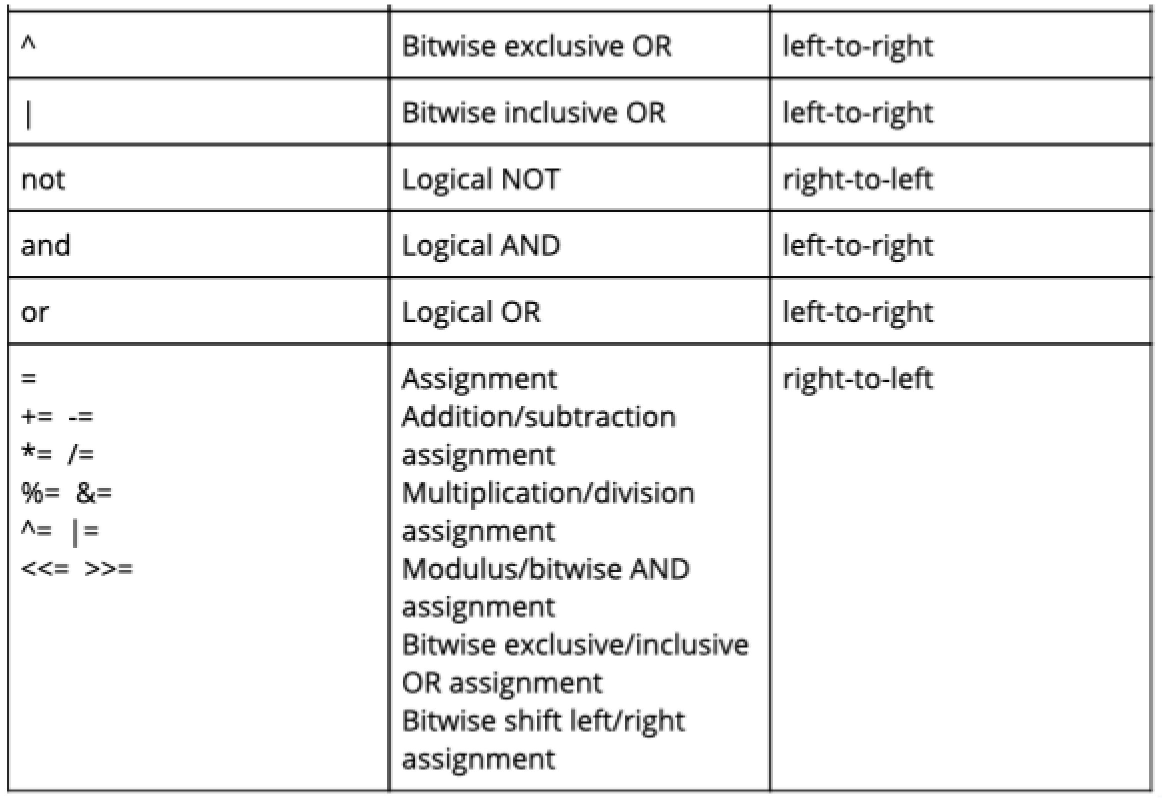
Previous

Next

Notes

Precedence and Associativity of Operators in Python

Operator precedence determines the order in which operations are processed.

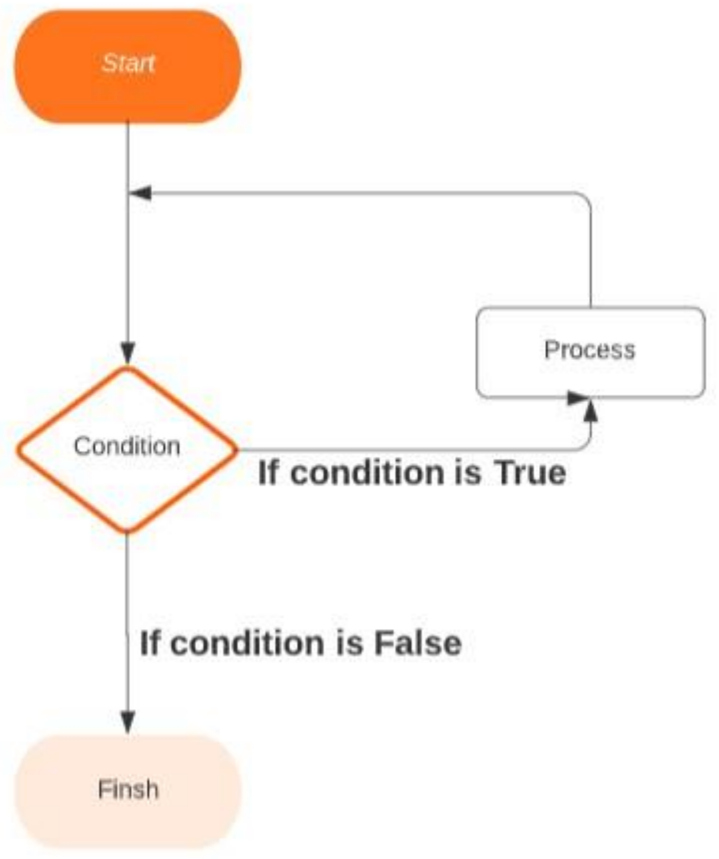
Operator	Description	Associativity
( )	Parentheses	left-to-right
**	Exponent	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
is, is not in, not in	Identity Membership operators	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
not	Logical NOT	right-to-left
and	Logical AND	left-to-right
or	Logical OR	left-to-right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

Introduction to Decision Making

Decision-making is required when we want to execute a code only if a certain condition is satisfied.

Decision structures evaluate multiple expressions that produce TRUE or FALSE as an outcome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision-making structure found in most of the programming languages −



Previous

Next

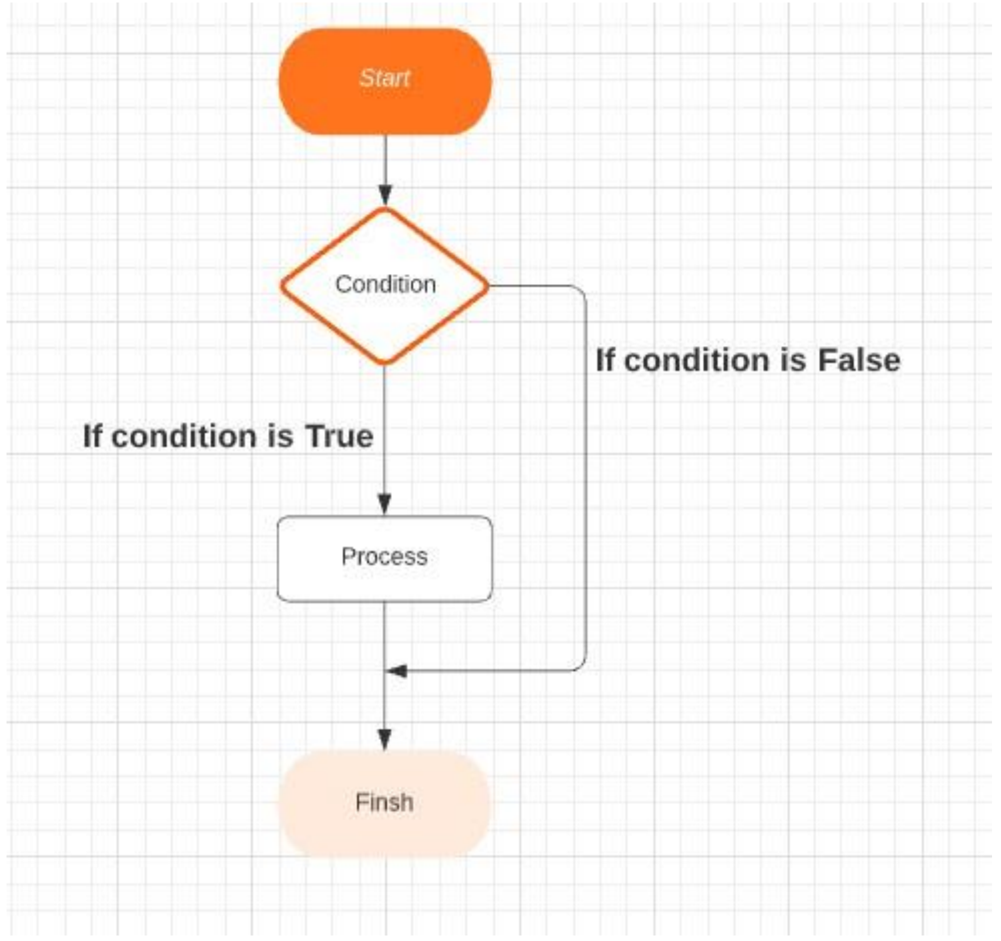
If Statement

Python if Statement Syntax

Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.

If the test expression is False, the statement(s) is not executed.

In Python, the body of the if the statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end. Python interprets non-zero values as True. None and 0 are interpreted as False.



Example:

```
#Check whether the given number is even or odd.
```

```
n = int(input())
if n % 2 == 0:
    print(n,"is even.")
```

output:

```
6
6 is even.
```

Previous

Next

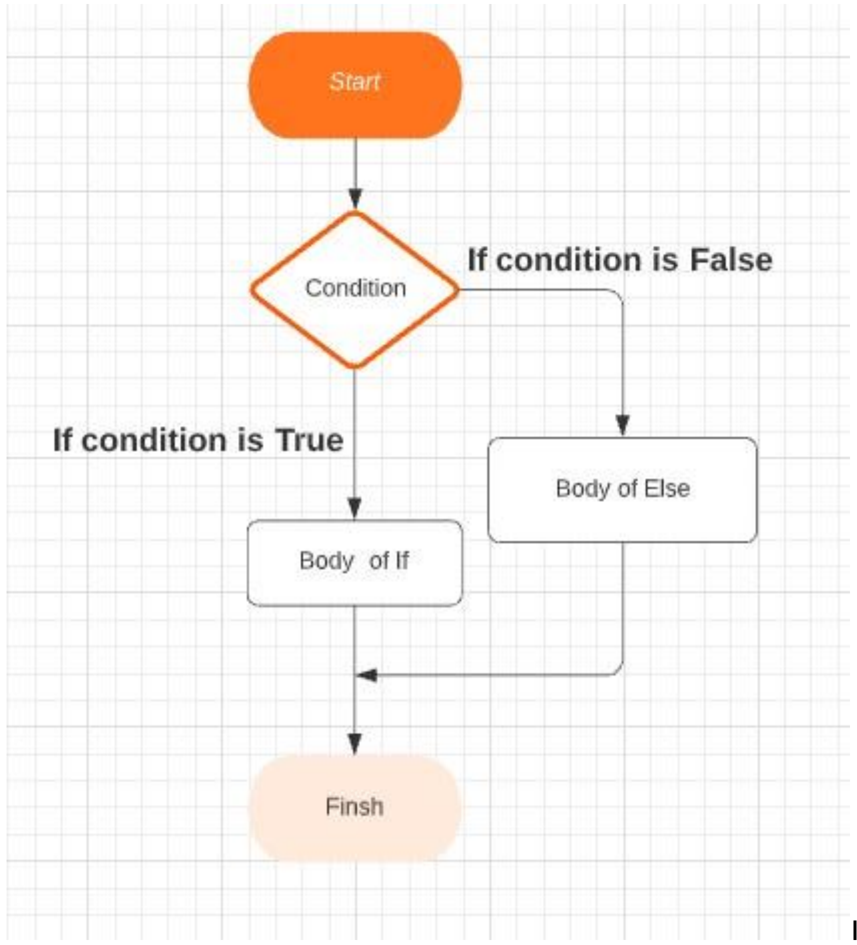
If else Statement

Python if...else Statement

```
if condition:
    Body of if
else:
    Body of else
```

The if..else statement evaluates the condition and will execute the body of if only when the condition is True.

If the condition is False, the body of else is executed. Indentation is used to separate the blocks.



If Else Flowchart

Example:

```
#Check whether the given number is even or odd.

n = int(input())
if n % 2 == 0:
    print(n,"is even.")
else:
    print(n,"is odd.")
```

output:

```
7
7 is odd.
```

In the above example if n is divisible by 2 means remainder will be zero then the condition will be true and “is even” will be output else “is odd” will be output.

Previous

Next

If..elif..else Statement

Python if...elif...else Statement

```
if condition1:
    Body of if

elif condition2:
    Body of elif

else:
    Body of else
```

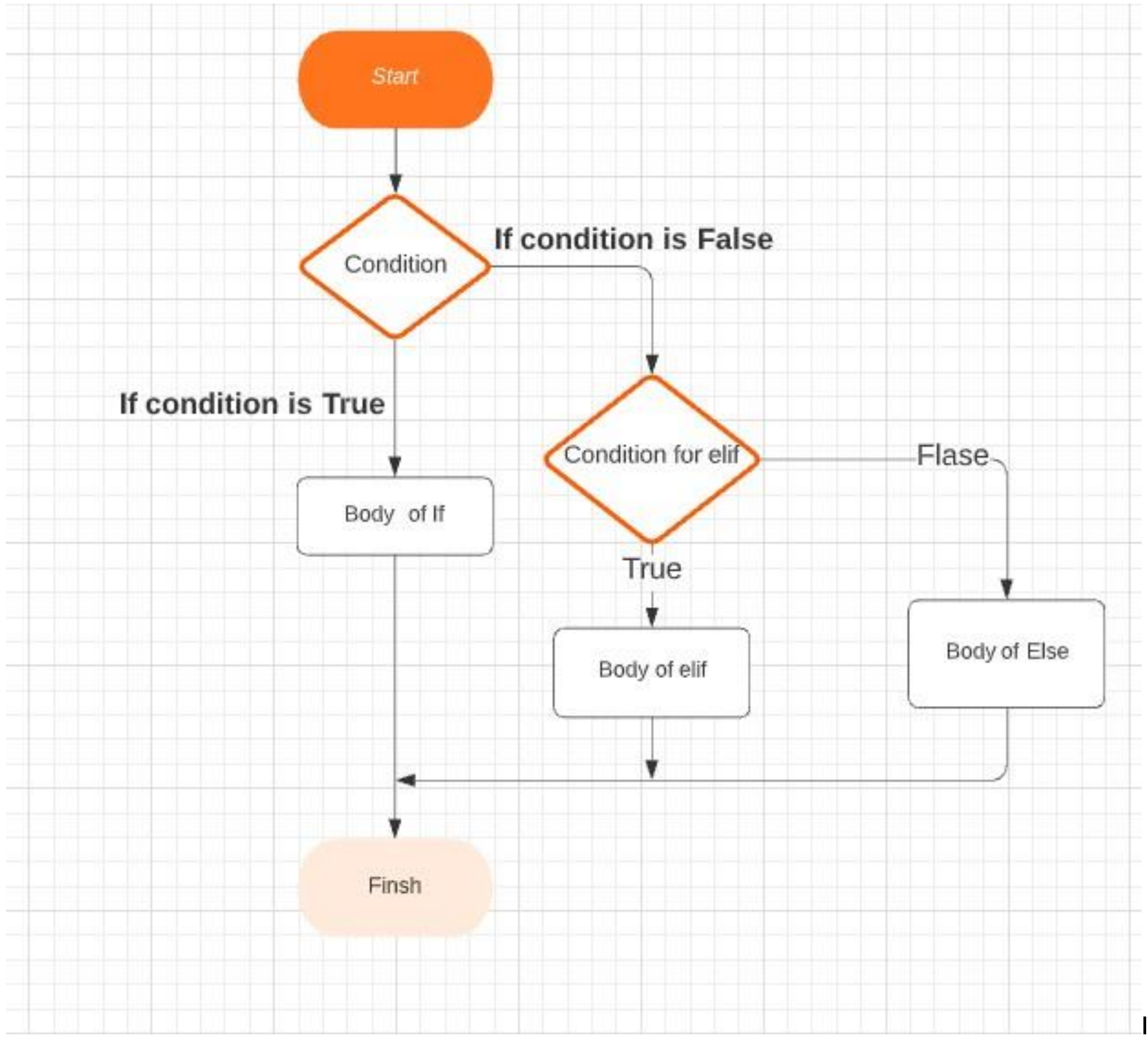
The elif is the same as else if in c++ or java. It allows us to check for multiple expressions.

If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, the body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if a block can have only one else block. But it can have multiple elif blocks.



If - elif - else

Example:

```
#find the largest among three number

a = 5
b = 6
c = 4

if a >= b and a >= c:
    print("a is greater")
```

```
elif b >= c and b >= a:
    print("b is greater")
    
else:
    print("c is greater")
```

output:

```
b is greater
```

Previous

Next

Introduction

Introduction to Loops

A 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 the keyword in other programming languages and works more like an iterative method as found in other object-orientated programming languages.

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

Previous

Next

for loop

Python for loop

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

Syntax of for Loop

```
for i in iterator:
    # statements
```

Here the iterable is a collection of objects like list, tuple. The indented statements inside the for loops are executed once for each item in an iterable. The variable var takes the value of the next item of the iterable each time through the loop.

Example:

```
#Print all numbers from 0 to n.

n = int(input())
for i in range(n + 1):
    print(i)
```

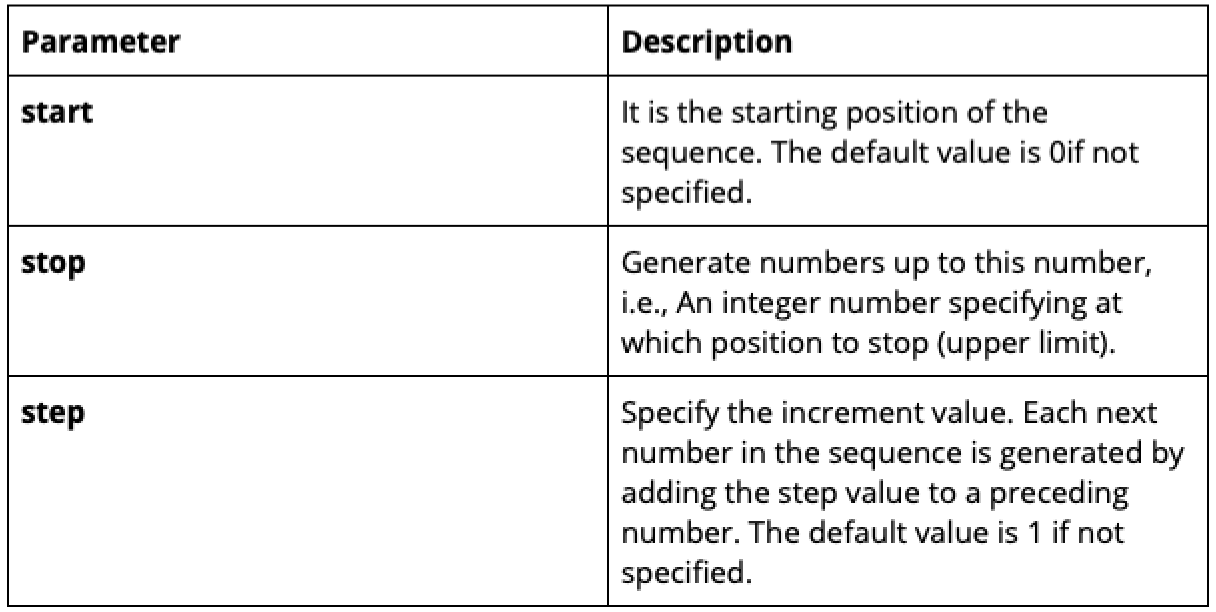
output:

```
5
0
1
2
3
4
5
```

Python range() function

Below is the syntax of the range() function.
range(start, stop,step)
It takes three arguments. Out of the three, two are optional. The start and step are the optional arguments.

Parameters



Parameter	Description
start	It is the starting position of the sequence. The default value is 0if not specified.
stop	Generate numbers up to this number, i.e., An integer number specifying at which position to stop (upper limit).
step	Specify the increment value. Each next number in the sequence is generated by adding the step value to a preceding number. The default value is 1 if not specified.

Points to remember

- The **range()** function only works with the integers, So all arguments must be integers. You can not use float numbers or any other data type as a start, stop, and step value.
- All three arguments can be positive or negative.
- The step value must not be zero. If a step=0, Python will raise a **ValueError** exception.

range(start, stop)
When you pass two arguments to the range(), it will generate integers starting from the start number to stop -1.

Example:

```
for i in range(10,15):
    print(i)
```

output:

```
10
11
12
13
14
```

Previous
Next
while loop

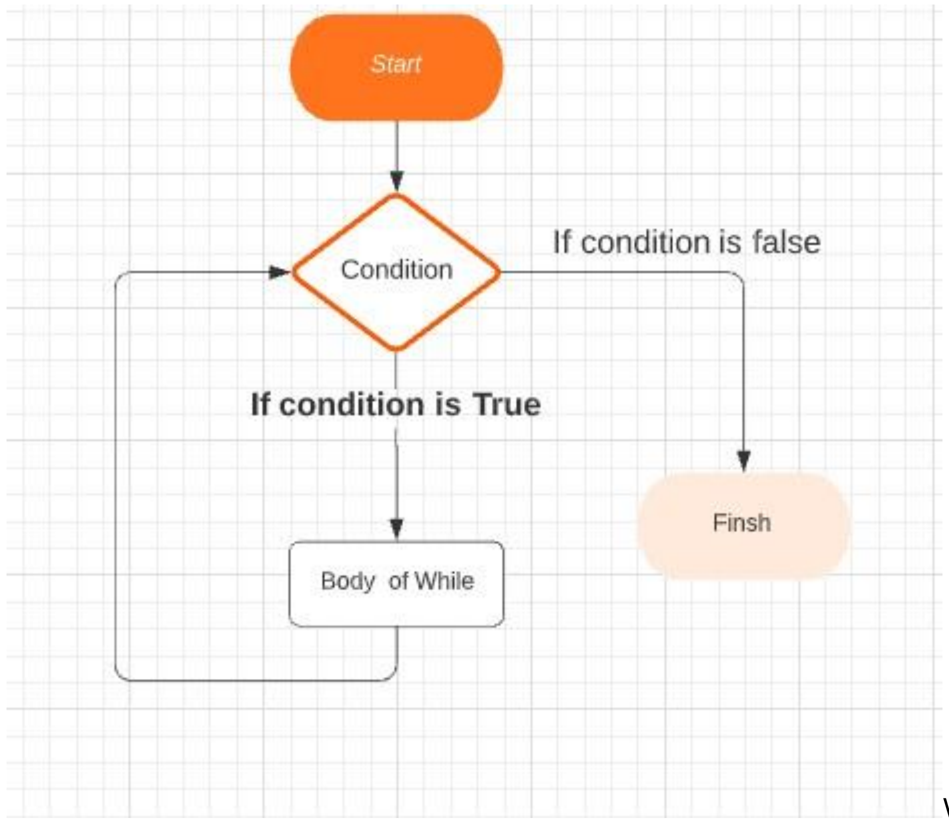
Python while loop

The while loop in Python is used to iterate over a block of code as long as the condition is true.

We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax of while Loop in Python

```
while condition:
    #body of a while
```



While loop flow chart

Example:

```
i = 0
while i < 5:
     print(i)
     i += 1
```

output:

```
0
1
2
3
4
```

Previous

Next

break Statement

Python break statement

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

Syntax:

```
break
```

Example:

```
i = 0

while i < 5:
    if i == 3:
        print("Executing 'break' statement in the next statment and the flow of execution is being", end = " ")
        break

    print(i)
    i += 1

print("shifted to here!")
```

output:

```
0
1
2
Executing 'break' statement in the next statment and the flow of execution is being shifted to here!
```

Previous

Next

continue Statement

Python continue statement

The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and for loops.

Syntax:

```
continue
```

Example:

```
i = 0
while i < 5:

    if i == 3:
        i += 1
        continue

    print(i)
    i += 1
```

output:

```
0
1
2
4
```

Previous

Next

pass statement

Python pass statement

The **pass** statement is a null statement. But the difference between **pass** and comment is that comment is ignored by the interpreter whereas **pass** is not ignored.

The **pass** statement is generally used as a placeholder i.e. when the user does not know what code to write. So user simply places **pass** at that line. Sometimes, **pass** is used when the user doesn't want any code to execute. So user simply places **pass** there as empty code is not allowed in loops, function definitions, class definitions, or in if statements. So using **pass** statement user avoids this error.

Syntax:

```
pass
```

Example:

```
i = 0
while i < 5:
    if i == 3:
        i += 1
        pass

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

output:

```
0
1
2
4
```

Previous

Next

return Statement

Python return statement

A **return** statement is used to end the execution of the function call and returns the result (value of the expression following the return keyword) to the caller. The statements after the **return** statements are not executed. If the **return** statement is without any expression, then the special value None is returned.

Note: Return statement can not be used outside the function.

Syntax:

```
def function():
    statements
    
    return [statement]
```

Example:

```
def fact(n):
    fact = 1
    for i in range(1,n+1):
        fact = fact * i
        
    return fact
    
ans = fact(5)
print(ans)
```

output:

```
120
```

Returning Multiple Values

Example:

```
def fun():


    a = 2
    b = 3
    return a,b


a,b = fun()
print(a,b)
```

output:

2 3

Previous

Next

Introduction

Introduction to functions

What Is A Function?

A Function is a sequence of statements/instructions that performs a particular task. A function is like a black box that can take certain input(s) as its parameters and can output the desired result(s) post performing some operations based on the input parameters**.** A function is created so that one can use a block of code as many times as needed just by using the name of the function.

Why Do We Need Functions?

- **Reusability:** Once a function is defined, it can be used over and over again. You can call the function as many times as it is needed. Suppose you are required to find out the area of a circle for 10 different radii. Now, you can either write the formula πr2 10 times or you can simply create a function that takes the value of the radius as an input and returns the area corresponding to that radius. This way you would not have to write the same code (formula) 10 times. You can simply invoke the function every time.

- **Neat code:** A code containing functions is concise and easy to read.

- **Modularisation:** Functions help in modularizing code. Modularization means dividing the code into smaller modules, each performing a specific task.

- **Easy Debugging:** It is easy to find and correct the error in a function as compared to raw code.

Previous

Next

How to Declare Function

Defining Functions In Python

A function, once defined, can be invoked as many times as needed by using its name, without having to rewrite its code.

A function in Python is defined as per the following syntax:

```
def <function-name>(<parameters>):
    """ Function's docstring """
    <Expressions/Statements/Instructions>
```

● Function blocks begin with the keyword **def** followed by the **function** name and **parentheses** (( )).

● The input **parameters** or **arguments** should be placed within these parentheses. You can also define parameters inside these parentheses.

● The first statement of a function is optional - the documentation string of the function or **docstring.** The **docstring** describes the functionality of a function.

● The code block within every function starts with a **colon** (:) and is **indented**. All statements within the same code block are at the same indentation level.

● The return statement exits a function, optionally passing back an expression/value to the function caller.

Example:

Let us define a function to add two numbers.

```
def add(a,b):

    return a + b
```

Output:

```
The above function returns the sum of two numbers a and b.
```

In the example given above, the sum a +b is returned.

Note: In Python, you need not specify the **return** type i.e. the **data type** of returned
value

Calling/Invoking A Function

Once you have defined a function, you can call it from another function, program, or even the Python prompt. To use a function that has been defined earlier, you need to write a **function call**.

A **function call** takes the following form:

```
<function-name> (<value-to-be-passed-as-argument>)
```

The function definition does not execute the function body. The function gets executed only when it is called or invoked. To call the above function we can write:

```
add(5,7)
```

In this function call, a = 5 and b = 7.

Arguments And Parameters

As you know that you can pass values to functions. For this, you define variables to receive values in the **function definition** and you send values via a function call statement. For example, in the **add()** function, we have variables **a** and **b** to receive the values and while calling the function we pass the values 5 and 7. We can define these two types of values:

- **Arguments:** The values being passed to the function from the function call statement are called arguments. Eg. 5 and 7 are arguments to the **add()** function.

- **Parameters:** The values received by the function as inputs are called parameters. Eg. **a** and **b** are the parameters of the **add()** function.

Previous

Next

Function Types

Types Of Functions

We can divide functions into the following two types:

1. **User-defined functions**: Functions that are defined by the users.
   Eg. The **add()** function we created.

Example:

```
#User define function add.
def add(a,b):

   #Inbuilt function print.
   print(a + b)


#Driver's code
z = add(2,3)
print(z)
```

output:

```
5
```

2. **Inbuilt Functions**: Functions that are inbuilt in python.
   Eg. The **print()** function.
   Eg. The **type()** function.

Example:

```
a = 5
string = "saif"

#Inbuilt print function
print(a)

#Inbuilt type function to check the DataType of variable.
print(type(string))
```

output:

```
5
<class 'str'>
```

Previous

Next

Function Overloading

Function Overloading:

As opposed to other languages (for example C++), method or function overloading is not supported in Python by default but that can be achieved in various ways.

We can defined multiple functions with the same name but only the last function will be considered, all the rest gets hidden.

Example:

Method 1 (Not The Most Efficient Method):

```
def mult(a, b):
    print(a * b)
```

```
def mult(a, b, c):

    print(a * b * c)


mult(2, 4, 3)
mult(2, 7)
```

output:

```
10
TypeError: mult() missing 1 required positional argument: 'c'
```

In the above code, there are two mult() methods, but the python compiler can only see the last i.e the one with 3 parameters. Therefore, even though we can define multiple methods with the same name and different arguments, but only the last method of them can be used. Calling any of the other methods will produce an error. Like here calling will **mult(2,3)** an error.

This issue can be overcomed by the following method:

Method 2 (Efficient One):
By Using Multiple Dispatch Decorator

Multiple Dispatch Decorator Can be installed by:
```
pip3 install multipledispatch
```

```
from multipledispatch import dispatch

@dispatch(int,int)
def mult(a, b):
    print(a * b)

@dispatch(int,int,int)
def mult(a, b, c):
    print(a * b * c)


add(2, 3, 5)
add(2, 3)
```

output:

```
10
5
```

Previous

Next

Variables Scope

Scope of Variables

All variables in a program may not be accessible at all locations in that program. Part(s) of the program within which the variable name is legal and accessible, is called the scope of the variable. A variable will only be visible to and accessible by the code blocks in its scope.

There are broadly two kinds of scopes in Python −

- Global scope
- Local scope

Global Scope :

A variable/name declared in the top-level segment **(__main__)** of a program is said to have a global scope and is usable inside the whole program (Can be accessed from anywhere in the program).

In Python, a variable declared outside a function is known as a global variable. This means that a global variable can be accessed from inside or outside of the function.

Creating a Global Variable

Consider the given code snippet:

```
x = "Global Variable"
def foo():
    print("Value of x: ", x)
foo()
```

Here, we created a global variable **x = "Global Variable"**. Then, we created a function foo to print the value of the global variable from inside the function. We get the output :

```
Global Variable
```

Thus we can conclude that we can access a global variable from inside any function.

What if you want to change the value of a Global Variable from inside a function?

Consider the code snippet:

```
x = "Global Variable"
def foo():
    print("Value of x: ", x)
foo()
```

In this code block, we tried to update the value of the global variable x. We get an output as:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

This happens because, when the command **x=x-1**, is interpreted, Python treats this **x** as a local variable and we have not defined any local variable **x** inside the function foo().

Local Scope:

Variables that are defined inside a function body have a local scope. This means that local variables can be accessed only inside the function in which they are declared.

Creating a Local Variable

We declare a local variable inside a function. Consider the given function definition:

```
def foo():
  y = "Local Variable"
  print(y)
foo()
```

We get the output as:

```
Local Variable
```

Accessing A Local Variable Outside The Scope

```
def foo():
  y = "local"
foo()
print(y)
```

In the above code, we declared a local variable y inside the function foo(), and then we tried to access it from outside the function. We get the output as:

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

We get an error because the lifetime of a **local variable** is the function it is defined in. Outside the function, the variable does not exist and cannot be accessed. In other words, a variable cannot be accessed outside its scope.

Previous

Next

Variable Length Arguments

Passing Variable-length arguments

In Python, a variable number of arguments can be passed to a function using special symbols. There are two special symbols:

Python uses these special Symbols for passing arguments:-

1.)*args (for Non-Keyword Arguments)
2.)**kwargs (for Keyword Arguments)

1.) *args

The special syntax *args in function definitions is used to pass a variable number of arguments to a function. It is used to pass a non-key-worded, variable-length argument list.

- *args allows us to take in more arguments than the number of formal arguments that we previously defined i.e any number of extra arguments can be tacked on to our current formal parameters (including zero extra arguments). This is because of the symbol * that is being used.
- For example, if we want to make an addition function that supports taking any number of arguments and able to add them all together. In this case we can use *args.
- The variable associated with the * becomes iterable and we can also run other higher-order functions such as map and filter, etc.

Example:

```
def sum(*args):

    sm = 0

    for i in args:
        sm += i

    print(sm)


```

```
sum(2,3,4,5)
```

output:

```
14
```

2.) **kwargs

The special syntax ****kwargs** in function definitions is used for passing a keyworded, variable-length argument list. We use double star here as it allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where we provide a name to the variable as we pass it into the function.
- kwargs is similar to a dictionary where each keyword is mapped to a value which we pass along with it. That is why on iterating over the contents of kwargs the elements do not seem to be stored in any specific order.

Example:

```
def fun(**kwargs):
    for key,value in  kwargs.items():
        print(key,value)


fun(name = "saif", roll = 101)
```

output:

```
name saif
roll 101
```

Previous
Next
Default Parameters

Python Default Parameters

Function parameters can have default values in Python. We can provide a default value to a parameter by using the assignment operator (=). Here is an example.

```
def greet(person, wish="Happy Birthday"):
    """This function prints the provided message in wish. If
    wish is empty, it defaults to "Happy Birthday" """
    print("Hello", person + ', ' + wish)

greet("Rita")
greet("Hardik", "Happy New Year")
```

output:

```
Hello Rita, Happy Birthday
Hello Hardik, Happy New Year
```

In this function, the parameter **person** does not have a default value and must be passed(mandatory) during a call.
On the other hand, the parameter **greet** has a default value of **"Happy Birthday"**. So, it is optional during a call. If an argument is passed corresponding to the parameter, it will overwrite the default value, otherwise, it will use the default value.

Important Points to be kept in mind while using default parameters:

- Any number of parameters in a function can have a default value.
- The conventional syntax for using default parameters states that once we have passed a default parameter, all the parameters to its right must also have default values.
- In other words, non-default parameters cannot follow default parameters.

For example: if we had defined the function header as:

```
def greet(wish = "Happy Birthday", name):
    #statement
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

Thus to summarise, in a function header, any parameter can have a default value unless all the parameters to their right have their default values.

Previous
Next
Introduction

Lists

Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Example:

```
>>> a = [] #Here we are creating onedimensional(1-D)List.
>>> a
```

output:

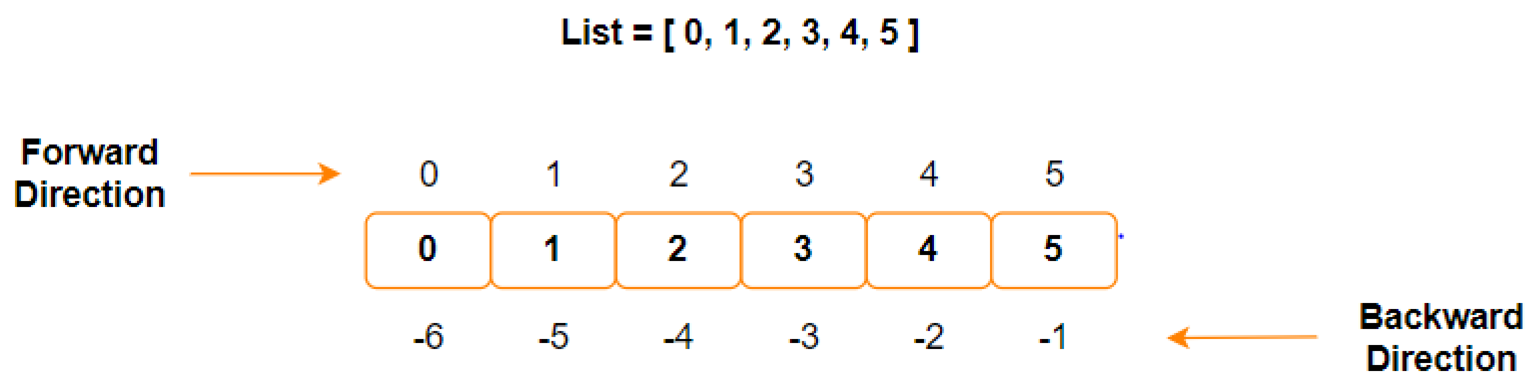
```
[]
```

Characteristics of Lists

The list has the following characteristics:

- Lists in python are ordered.
- The elements of a list can be accessed by their indices.
- Lists are mutable.
- A list can store a number of various/different elements.

List indexing



Example:

```
list = [0,1,2,3,4,5]
print(list[0])
print(list[2])
print(list[-1])
```

output:

```
0
```

```
2
5
```

Previous
Next
One Dimensional(1-D)List

How to create a One-Dimensional(1-D) list?

In Python programming, we can define an array i.e. One-Dimensional(1-D) list by placing all the items (elements) inside **square brackets []**, and all elements are separated by commas. One of the most powerful features of a Python list is that it may have different types (integer, float, string, etc.) of elements in a single list.

The syntax **for declaring a one dimension list:**

```
l = []
l = list()
```

Example:

```
# Declaring a empty list.
oneDList = []
print(oneDList)

# List of integers.
myList = [1, 2, 3]
print(myList)

# Declaring a list that contains multiple datatypes.
myList = [1, "Hello", 3.14]
print(myList)
```

Output:

```
[]
[1, 2, 3]
[1, 'Hello', 3.14]
```

In python, a list can also have another list as an item.

```
# Declaring a Nested list.
```

```
myList = ["hello", [2, 4, 6], ['a']]
```

Access List Elements

There are multiple ways in which we can access the elements of a list.

List Index

We can use the index operator **[]** to access an element in a list. In Python, indexing starts with 0 i.e index of the first element is 0. So, a list having 5 elements will have an index from 0 to 4.

If we try to access indices out of the range then python will raise an **IndexError**. Also, the index must be an integer else python will give a type error.

Nested lists are accessed using nested indexing.

```
# List indexing
myList = ['h', 'e', 'l', 'l', 'o']

print(myList[0])

print(myList[1])

print(myList[4])

# Nested List
nList = ["ninja", [0, 0, 1, 1]]

# Nested indexing
print(nList[0][1])
print(nList[1][3])

# Error! The only integer that can be used for indexing
print(myList[4.0])
```

Output:

```
h
e
o
i
1
```

```
Traceback (most recent call last):
  File "<string>", line 21, in <module>
TypeError: list indices must be integers or slices, not float
```

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.

Example:

```
# Negative indexing in lists

my_list = ['n','i','n','j','a']
print(myList[-1])
print(myList[-5])
```

Output:

```
a
n
```

How to slice lists in Python?

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

With the ':' operator, we can specify the start and end index for the slicing and the index jump in slicing. List slicing returns a new list from the existing list. Note that last the end index is not included in slicing i.e the range of slicing is [initial, end).

Syntax:

```
Lst[ Initial: End: IndexJump ]
```

Example:

```
# List slicing in Python
myList = ['h','e','l','l','o','n','i','n','j','a','s']

# Elements 3rd to 5th
print(myList[2:5])

# Elements from beginning to 4th index.
```

```
print(myList[:-5])

# Elements 6th index to end.
print(myList[5:])

# Elements from beginning to end.
print(myList[:])
```

Output:

```
['l', 'l', 'o']
['h', 'e', 'l', 'l', 'o', 'n']
['n', 'i', 'n', 'j', 'a', 's']
['h', 'e', 'l', 'l', 'o', 'n', 'i', 'n', 'j', 'a', 's']
```

Add/Change List Elements

In python, Lists are mutable, i.e. their elements can be changed, unlike- strings or tuples.
We can use the assignment operator '=' to change an item or a range of items.

Example:

```
# Change
myList = [1, 3, 6, 8]

# Change the 3rd elemet to 5
odd[2] = 5
print(odd)
```

Output:

```
[1, 4, 5, 8]
```

We can add one item to a list using the '**append()'** method or add several items using the **extend()** method.

Example:

```
# Appending and Extending lists in Python

odd = [1, 3, 5]
```

```
odd.append(7)
print(odd)
odd.extend([9, 11, 13])
print(odd)
```

Output:

```
[1, 3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

We can also use the **+** operator to combine two lists. This is also called concatenation.
The ***** operator repeats a list for the given number of times.

Example:

```
# Concatenating and repeating lists

myList = [1, 3, 5]
print(myList + [20, 2, 6])

print(["ab"] * 3)
```

Output:

```
[1, 3, 5, 20, 2, 6]
['ab', 'ab', 'ab']
```

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.

Example:

```
# Deleting list items
myList = ['n', 'i', 'n', 'j', 'a']

# We can delete one item as
del myList[2]
print(myList)
```

```
# Deleting multiple items in list
del myList[1:3]
print(myList)

# Deleting an entire list
del myList

# Error: List not defined
print(myList)
```

Output:

```
['n', 'i', 'j', 'a']
['n', 'a']
Traceback (most recent call last):
  File "<string>", line 18, in <module>
NameError: name 'myList' is not defined
```

We can use the **remove()** method to remove the given item or the **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. This helps us implement lists as stacks (first in, last out data structure).
We can also use the **clear()** method to empty a list.

Example:

```
myList = ['n', 'i', 'n', 'j', 'a']
myList.remove('i')

print(myList)

print(myList.pop())

print(myList)

myList.clear()

# Output: []
print(myList)
```

Output:

```
['n', 'n', 'j', 'a']
```

```
a
['n', 'n', 'j']
[]
```

Previous

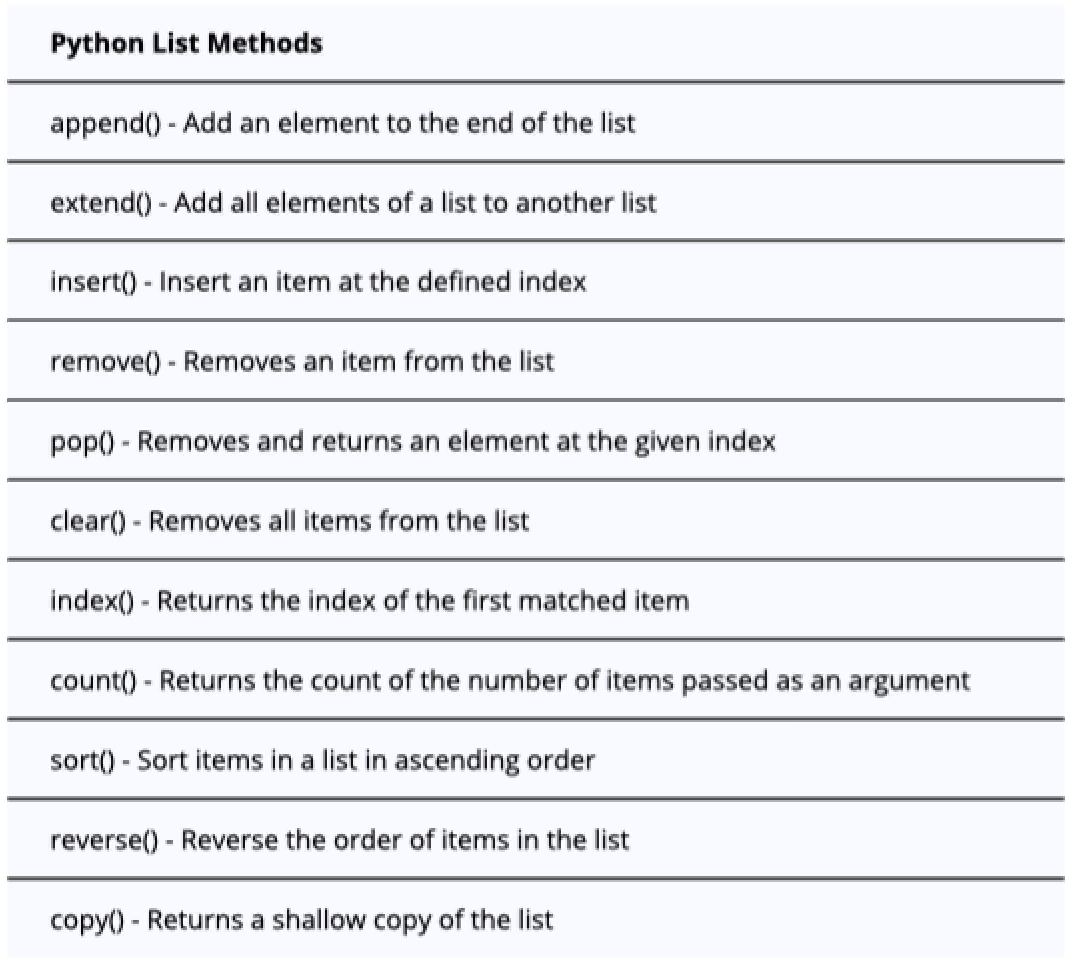
Next

List Methods

List Methods

Methods that are available with list objects in Python programming are tabulated below.
They have accessed as **list.method()**. Some of the methods have already been used above.

Some examples of Python list methods:



Python List Methods
append() - Add an element to the end of the list
extend() - Add all elements of a list to another list
insert() - Insert an item at the defined index
remove() - Removes an item from the list
pop() - Removes and returns an element at the given index
clear() - Removes all items from the list
index() - Returns the index of the first matched item
count() - Returns the count of the number of items passed as an argument
sort() - Sort items in a list in ascending order
reverse() - Reverse the order of items in the list
copy() - Returns a shallow copy of the list

Example:

```
# Python list methods
my_list = [3, 8, 1, 6, 0, 8, 4]

print(my_list.index(8))
# Output: 1


print(my_list.count(8))
# Output: 2

my_list.sort()
print(my_list)
# Output: [0, 1, 3, 4, 6, 8, 8]

my_list.reverse()
print(my_list)
# Output: [8, 8, 6, 4, 3, 1, 0]
```

Output:

```
1
2
[0, 1, 3, 4, 6, 8, 8]
[8, 8, 6, 4, 3, 1, 0]
```

Passing a List as an Argument

You can send any data type of argument to a function (string, list, etc.), and it will be treated as the same data type inside the function.
E.g. if you send a list as an argument, it will still be a List when it reaches the function:

Example:

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

```
my_function(fruits)
```

Output:

```
apple
banana
cherry
```

Previous

Next

Notes

List Comprehension

In Python, list comprehension is a powerful, elegant, and concise way to create a new list from an existing list.
A list comprehension consists of an expression followed by **for** statement inside square brackets.

Syntax:

```
newList = ['expression' for 'item' in iterable]
```

Let's take an example when we are required to find '2 ^ i' for each ‘i’ from 1 to 10, list comprehension can be used as-

Example:

```
pow2 = [2 ** x for x in range(10)]
print(pow2)
```

Output:

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

The above code is a simple one-line code for the below code.

```
pow2 = []
for x in range(10):
   pow2.append(2 ** x)
```

Similarly, we can use other expressions to modify or create a new list.

Example:

```
newList = [x for x in range(10) if x % 2 == 1]
print(newList)
```

Output:

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

Previous

Next

Two Dimensional(2-D) List

Two-Dimensional(2-D) Lists

Introduction

- A Two-Dimensional(2-D) is an array of arrays.
- It represents a matrix with rows and columns of data.
- We can get particular data from the 2-D list by its row index and column index.
- Consider an example of recording test scores of 4 students, in 4 subjects Such data can be presented as a two-dimensional list as below.

```
student 1 - 9, 8, 5, 2
student 2 - 5, 6, 10, 6
student 3 - 10, 8, 5, 5
student 4 - 10, 10, 8, 6
```

The above data can be represented as a Two-Dimensional(2-D) list as below, where each row represents a student and columns represents a subject.

```
twoDList = [[9, 8, 5, 2 ], [5, 6, 10, 6 ], [10, 8, 5, 5 ], [10, 10, 8, 6]]
```

Accessing Values in a Two-Dimensional(2-D) list

The data elements in Two-Dimensional(2-D)Lists can be accessed using two indices. One index referring to the main or parent list (inner list) and another index referring to the position of the data element in the inner list.

The example below illustrates how it works.

```
twoDList = [[9, 8, 5, 2 ], [5, 6, 10, 6 ], [10, 8, 5, 5 ], [10, 10, 8, 6]]
```

```
# List at index 0 in twoDList
print(twoDList[0])

# Element at index 2 in list at index 2 in twoDList
print(twoDList[2][2])
```

When the above code is executed, it produces the following result −

```
[9, 8, 5, 2 ]
5
```

Input Of Two-Dimensional(2-D) Lists

We will discuss two common ways of taking user input:

1. **Line Separated Input**- Different rows in different lines.
2. **Space Separated Input**- aking input in a single line.

Line Separated Input:

We will use list comprehension to take user input. For every row, we take a list as input. The syntax would be:

```
# Number of rows
n = int(input())

input= [[int(col) for col in input.split()] for row in range n]

print(input)
```

User Input:

```
4
9 8 5 2
5 6 10 6
10 8 5 5
10 10 8 6
```

Output:

```
[[9, 8, 5, 2 ], [5, 6, 10, 6 ], [10, 8, 5, 5 ], [10, 10, 8, 6]]
```

Space Separated Input:

For this we will require the user to specify the number of rows (say **row**) and columns (say **col**) of the Two-Dimensional(2-D) list. The user will then enter all the elements of the Two-Dimensional(2-D) list in a single line. We will first convert this input stream to a list (say **inputL**). The next step would be using list comprehension to make a Two-Dimensional(2-D) list (say **fnlList**) from this list. It can be observed that **fnlList[i,j] = inputL[i*row + j]**. The python code for this can be written as follows:

```
#Number of rows
row = int(input())

#Number of columns
col = int(input())

#Converting input string to list
inputL= input().split()

fnlList=[[int(inputL[i*col+j]) for j in range(col)] for i in range(row)]

print(fnlList)
```

User Input:

```
4
4
9 8 5 2 5 6 10 6 10 8 5 5 10 10 8 6
```

Output:

```
[[9, 8, 5, 2 ], [5, 6, 10, 6 ], [10, 8, 5, 5 ], [10, 10, 8, 6]]
```

Printing in Multiple Lines Like a Matrix

We can use nested **for** loops to print a 2D List. The outer loop iterates over the rows and the inner loop will iterate over columns.
Example:

```
twoDList = [[9, 8, 5, 2 ], [5, 6, 10, 6 ], [10, 8, 5, 5 ], [10, 10, 8, 6]]

# Iterate in row of 2-D list
for row in T:
        # Iterate in coloumn in 2-D list.
    for col in row:
        print(col, end=" ")
    print()
```

Output:

```
9 8 5 2
5 6 10 6
10 8 5 5
10 10 8 6
```

Previous
Next
Jagged Lists

Jagged Lists

Till now we have seen lists with an equal number of columns in all rows. Jagged lists are two-dimensional lists with a variable number of columns in various rows. Various operations can be performed on such lists, the same way as in 2D lists with the same number of columns throughout. Some examples of jagged lists are shown below:

```
T1 = [[11,12,5,2],[10,6],[10,8,12],[12]]
T2 = [[1,2,3], 1, 3, [1,2]]
T3 = [[1],[],[1,2,1],0]
```

The elements in a jagged list can be accessed the same way as shown:

```
>>> T1[3]
[12]
>>> T1[0][2]
5
```

Note: Keep in mind the range of columns for various rows. Indices out of the range can produce **indexOutOfRange** error.

Previous

Next

String Creation

Introduction

● A character is simply a symbol. For example, the English language has 26 characters;
while a string is a contiguous sequence of characters.

● In Python, a string is a sequence of Unicode characters.

● Python strings are immutable, which means they cannot be altered after they are created.

Note: Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn more about Unicode from Python Unicode.

Given below are some examples of Unicode Encoding

```
string ='pythön!'→ Default UTF-8 Encoding: b'pyth\xc3\xb6n!'
string ='python!'→ Default UTF-8 Encoding: b'python!'
```

How to create a string in Python?

Strings in Python can be created using single quotes, double quotes and even triple quotes.

Example:

```
# Defining strings in Python

#Using single quotes
s1 = 'Hello'
print(s1)

#Using double quotes
s2 = "Hello"
print(s2)

print(type(s1))
print(type(s2))
```

Output:

```
Hello
Hello
<class 'str'>
<class 'str'>
```

Note:
We can use triple quotes to create docstrings and/or multiline strings.

Example:

```
s = '''
Hey, may name is Afzal
I am 27 years old.
I work as a Project Manager at Coding Ninjas.
'''


#Multiline string
print(s3)
```

Output:

```
Hey, may name is Afzal
I am 27 years old.
I work as a Project Manager at Coding Ninjas.
```

If you wish to print a string in a new line you can use the new line character **'\n'**. This is used within the strings. For example:

```
print('First line\nSecond line')
```

This would yield the result:

```
First Line
Second Line
```

Accessing Characters in a String

String indices start at 0 and go on till 1 less than the length of the string. We can use the index operator [ ] to access a particular character in a string. Eg.

Index:	**0**	**1**	**2**	**3**	**4**
String "hello":	**h**	**e**	**l**	**l**	**o**

Note: Trying to access indexes out of the range (0, lengthOfString-1), will raise an **IndexError**. Also, the index must be an integer. We can't use float or other data types; this will result in **TypeError**.

Let us take an example to understand how to access characters in a String:

```
s= "hello"

print(s[0]) #Output: h
print(s[2]) #Output: l
print(s[4]) #Output: o
```

Negative Indexing

Python allows negative indexing for strings. The index of **-1** refers to the last character, **-2** to the second last character, and so on. The negative indexing starts from the last character in the string.

Positive Indexing:	**0**	**1**	**2**	**3**	**4**
String "hello":	**h**	**e**	**l**	**l**	**o**
Negative Indexing:	**-5**	**-4**	**-3**	**-2**	**-1**

Let us take an example to understand how to access characters using negative indexing in a string:

```
s= "hello"

print(s[-1]) #Output: 0
print(s[-2]) #Output: l
print(s[-3]) #Output: l
```

Previous

Next

String Concatenation

Python String Concatenation

String Concatenation is the technique of combining two strings. String Concatenation can be done using many ways.

We can perform string concatenation using the following ways:

1. Using + operator
2. Using join() method
3. Using % operator
4. Using format() function

Using + Operator

The process of combining strings is called string concatenation. In Python, strings can be concatenated in the following ways:

1. Using + operator
2. Using join() method
3. Using % operator
4. Using format() function

Using + Operator

The + operator functions similar to the arithmetic + operator, but here both the operands must be string. It is one of the most easy ways to concatenate strings.

Note: Strings are immutable objects, hence the result of concatenation needs to be stored in a new variable(string object).

```
# Python program to demonstrate
# string concatenation

var1 = "Coding "
var2 = "Ninjas"

# + Operator is used to combine strings
var3 = var1 + var2

print(var3)
```

Output:

```
Hello World
```

Using join() Method

The ***join()*** string method returns a string by joining all the character elements of an iterable, separated by a string separator.

The join() method provides a very convenient way to join characters of an iterable(such as list, tuple or string objects) such that the elements are joined by a string separator, and at the end return the concatenated string.

Syntax:

```
string.join(iterable)
```

Parameters: The join() method takes iterable – objects capable of returning their members one at a time. Some examples are **List, Tuple, String, Dictionary and Set.**

Return Value: The join() method returns a string concatenated with the elements of iterable.

Type Error: If the iterable contains any non-string values, it raises a TypeError exception.

Example:

```
# Python program to demonstrate string concatenation

var1 = "Coding"
var2 = "Ninjas"

# join() method is used to join strings with an empty character as a separator("")
var3 = "".join([var1, var2])
print(var3)


# This time we use space(" ") as a separator for join() method.
var3 = " ".join([var1, var2])
print(var3)


li = ['One', 'Two', 'Three']
sep = '-Separator-'
print(sep.join(li))


var = "TESTING"
sep = "*"
print(sep.join(var)) # String is also an iterable as it is a collection of characters.
```

Output:

```
CodingNinjas
Coding Ninjas
```

```
One-Separator-Two-Separator-Three
T*E*S*T*I*N*G
```

In the above examples initially var1 stores "Coding" and var2 stores "Ninjas". First, we joined var1 and var2 with an empty character in between them which resulted in "CodingNinjas" and later we joined them with space as a separator. The join method accepts only lists as its argument whose size can be variable.

Using % Operator

We have used % operator for string formatting, but it can also be used for string concatenation. % operator helps both in string concatenation and string formatting.

Here is an example code to demonstrate the use of % for concatenation:
Example:

```python
# Python program to demonstrate
# string concatenation

var1 = "Coding"
var2 = "Ninjas"

# % Operator is used here to combine the string stored in var1 and var2
print("%s %s" % (var1, var2))
```

Output:

```
Coding Ninjas
```

Here, the % Operator combines the string that is stored in var1 and var2. The %s denotes string data type. The value in both the variables is passed to the string %s and becomes “Coding Ninjas”.

Using format() function

With Python 3.0, another function, the format() method has been introduced in the python library for handling string formatting which are more complex to be solved by % operator, in an elegant and efficient way.. format() method of the built-in string class provides a solution for complex variable substitutions and value formatting.

The syntax for format() method is:

string.format(var1, var2, var3,…)

Using a Single Formatter :

In Python, formatters work by employing replacement fields and different placeholders defined by a pair of curly braces {}, one curly brace for each string, and then calling {}.format(). The value we wish to concatenate is passed as function parameters to format().

```
Syntax : { } .format(value)
Parameters : 
(value) : Can be an integer, floating point, string, characters or even variables containing other values.

Returntype : Returns a formatted string with the value passed as parameter in the placeholder position.
```

Example:

```
# Python3 program to demonstarte
# the str.format() method

# using format option in a simple string
print ("{}, is best for DSA content." .format("CodingNinjas"))

# using format option for a
# value stored in a variable
str = "This article is provided at {}"
print (str.format("CodeStudio"))

# formatting a string using a numeric constant
print ("Hello, I am {} years old".format(20))
```

output:

```
CodingNinjas, is best for DSA content.
This article is provided at CodeStudio
Hello, I am 20 years old
```

Using Multiple Formatters :

Similar to what we did in the previous example, we can add more than one placeholders to insert in a string, Python automatically assigns each value to the placeholders in the order they appear. The first value is inserted in the first placeholder's position, second value is inserted in the second placeholder's position and so on.

```
Syntax : { } { } .format(value1, value2)

Parameters :
(value1, value2) : Similar to what we saw in single formatters, the values can be integers, floating point, numeric constants, strings, characters and even variables. The
difference lies that instead of one value we can pass multiple values this time.
```

```
Errors and Exceptions : 
IndexError : Occurs when the number of placeholders and the number of values passed do not match. Python usually assignes each placeholders with default index in order like 0, 1, 
2... Therefore when it encounters a placeholder but cannot find a value for it, it throws IndexError.
```

Example:

```
# Python program to demonstrate
# string concatenation

var1 = "Coding"
var2 = "Ninjas"

# format function is used here to
# combine the string
print("{} {}".format(var1, var2))
```

Output:

```
Coding Ninjas
```

Here, the format() function combines the strings stored in the var1 and var2. The curly braces {} depict the positions of the string in the result. The first curly braces is the position of the first string(var1), the second braces is the position of the second variable and so on. Finally, it prints the value “Coding Ninjas”.

Previous

Next

String Methods

Repeating/Replicating Strings

You can use the ***** operator to replicate a string specified number of times. Consider the example given below:

```
>>> s= "Ninja"
>>> print(s*3) #s*3 will produce a new string with s repeated thrice

NinjaNinjaNinja #Output
```

Note: You cannot use the ***** operator between two strings i.e. you cannot multiply a string by a different string.

String Slicing in Python

As the name suggests, Python slicing slices the string from the front to back to get a substring out of it.

Python supports slicing through two methods:

- slice() Constructor
- Extending Indexing

slice() Constructor

The slice() constructor creates a slice object representing the set of indices specified by range(start, stop, step)

Syntax:

```
slice= <String Name>[StartIndex : StopIndex : Steps]
Syntax:

slice(stop)
slice(start, stop, step)

Parameters:
start: Starting index from where we want the substring to start.
stop: Ending index where the substring needs to stop.
step: This is an optional argument that signifies the jump between each selected index.

Return Type: Returns a sliced object containing elements in the given range only.
```

Example:

```
# Python program to demonstrate
# String slicing

String ='CODINGNINJAS'
# Using slice constructor
s1 = slice(5)
s2 = slice(1, 5, 2)
s3 = slice(-1, -12, -2)

print("String slicing")
print(String[s1])
print(String[s2])
print(String[s3])
```

Output:

```
String slicing
CODIN
OI
SJIGIO
```

Extending indexing

Python can also use the syntax for indexing in place of slice object. This is an easy way to slice a string both syntax-wise and execution-wise.

Syntax:

```
string[start:end:step]
```

start, end, and step have the same mechanism as slice(), constructor.

Example:

```
# Python program to demonstrate
# string slicing
# String slicing

String ='ASTRING'
# Using indexing sequence
print(String[:3])
print(String[1:5:2])
print(String[-1:-12:-2])

# Prints string in reverse
print("\nReverse String")
print(String[::-1])
```

Output:

```
AST
SR
GITA
Reverse String
GNIRTSA
```

`GITA`

Comparing Strings

1.In string comparison, we aim to identify whether two strings are equivalent to each other and if not, which one is greater.
2.String comparison in Python takes place character by character i.e. each character in the same index, are compared with each other.
3.If the characters fulfill the given comparison condition, it moves to the characters in the next position. Otherwise, it merely returns **False**.

Note: Some points to remember when using string comparison operators:

●The comparisons are **case-sensitive**, hence the **same** letters in different
letter cases(upper/lower) will be treated as different characters.
●If two characters are different, then their Unicode value is compared; the character with the smaller Unicode value is considered to be lower.

The following operators are used for comparison:

● **==:** checks whether two strings are equal
● **!=:** checks if two strings are not equal
● **<:** checks if the left operand is smaller than the right.
● **>:** checks if the right operand is smaller than the left operand.
● **<=:** checks if the left operand is smaller than or equal to the right operand.
● **>=:** checks if the right operand is smaller than or equal to the left operand.

Example:

```
print("Ninja" == "Ninja")
print("Ninja" < "Ninja")
print("Ninja" > "Ninja")
print("Ninja" != "Ninja")
```

Output:

```
True
True
False
False
```

Using is and is not

The == operator checks if the value of both the left and right operands is the same or not whereas **is** operator checks if both the left and right operand refer to the same object. Similar explanation goes for != and **is not**.
Let us look at the following example:

Example:

```
str1 = "Ninja"
str2 = "Ninja"
str3 = str1

print("ID of str1 =", hex(id(str1)))
print("ID of str2 =", hex(id(str2)))
print("ID of str3 =", hex(id(str3)))
print(str1 is str1)
print(str1 is str2)
print(str1 is str3)

str1 += "s"
str4 = "Ninja"

print("\nID of changed str1 =", hex(id(str1)))
print("ID of str4 =", hex(id(str4)))
print(str1 is str4)
```

Output:

```
ID of str1 = 0x7fe6b544fdf0
ID of str2 = 0x7fe6b544fdf0
ID of str3 = 0x7fe6b544fdf0
True
True
True
ID of changed str1 = 0x7fe6b5417e30
ID of str4 = 0x7fe6b544fdf0
False
```

The object ID of the strings depends on the platform they are being checked and can vary for different machines. The object IDs of str1, str2, and str3 were the same; therefore, their comparison result is True in all the cases. After the object id of str1 is changed, the comparison of str1 and str2 will be false. Even after creation of str4, which has the same contents as in the new str1, the comparison will still be false as their object IDs are different.

Previous
Next

Notes

Iterating On Strings

There are multiple ways to iterate over a string in Python.

Using for loop

Example:

```
s = "MdAzmatAli"

# Using for loop
for i in s:
    print(i) #Print the character in the string
```

Output:

```
M
d
A
z
m
a
t
A
l
i
```

Using for loop and range()

Example:

```
s = "MdAzmatAli"

# Using for loop
for i in range(len(s)):
    print(s[i]) #Print the character in the string
```

Output:

Note: You can even use while() loops. Try using the while() loops on your own.

Previous

Next

Introduction

Introduction to OOPS

As the name suggests, Object-Oriented programming or OOPs refers to the language that uses the concept of class and object in programming. The main aim of OOPs is to implement real-world entities such as polymorphism, inheritance, encapsulation, abstraction, etc. The popular object-oriented programming languages are c++, java, python, PHP, c#, etc. ***Simula*** is the first object-oriented programming language.

Why do we use object-oriented programming?

- OOPs, make the development and maintenance of projects more manageable.
- OOPs, provide the feature of data hiding that is good for security concerns.
- We can provide the solution to real-world problems if we are using object-oriented programming.

OOPs Concepts: Let’s look at the topics which we are going to discuss.

- Access modifiers
- Class
- Object
- Encapsulation
- Abstraction
- Polymorphism

- Inheritance
- Constructor
- Destructor

What is an Object?

- The object is an entity that has a state and behavior associated with it. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
- Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number **12** is an object, the string **" Hello, world"** is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

What is a Class?

- A class is a **blueprint** (a plan basically) that is used to define (bind together) a set of variables and methods (Characteristics) that are common to all objects of a particular kind.

Example: If **Car** is a class, then **Maruti 800** is an object of the **Car** class. All cars share similar features like wheels,1steeringwheel, windows, breaks, etc.Maruti800(the **Car** object) has all these features.

Classes vs Objects (Or Instances)

- Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.
- In this module, you'll create a **Car** class that stores some information about the characteristics and behaviors that an individual **Car** can have.
- A class is a blueprint for how something should be defined. It doesn't contain any data. The **Car** class specifies that a name and a top-speed are necessary for defining a **Car**, but it doesn't contain the name or top-speed of any specific **Car**.
- While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the **Car** class is not a blueprint anymore. It's an actual car with a **name**, like Creta, and with a **top speed** of 200 Km/Hr.
- Put another way, a class is like a form or a questionnaire. An **instance** is like a form that has been filled out with information. Just like many people can fill out the same form with their unique information, many instances can be created from a single class.

Defining a Class in Python

- All class definitions start with the **class** keyword, which is followed by the name of the class and a colon(:). Any code that is indented below the class definition is considered part of the class's body.
- Here is an example of a **Car** class:

```
class Car:
    pass
```

- The body of the **Car** class consists of a single statement: the **pass keyword**. As we have discussed earlier, the **pass** is often used as a placeholder indicating where the code will eventually go. It allows you to run this code without Python throwing an error.
- **Note**: Python class names are written in CapitalizedWords notation by convention. **For example**, a class for a specific model of Car like the Bugatti Veyron would be written as **BugattiVeyron**. The first letter is capitalized. This is just a good programming practice.

- The **Car** class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Car objects should have. There are several properties that we can choose from, including **color**, **brand**, and **top-speed**. To keep things simple, we'll just use **color** and **top-speed**.

Example:

```
class Car:
   pass

c = Car
c.name = "ferrari"
c.topspeed = 400
print(c.name)
print(c.topspeed)
```

output:

```
ferrari
400
```

Previous

Next

Constructor

Constructor

- Constructors are generally used for instantiating an object.
- The task of a constructor is to initialize(assign values) to the data member of the class when an object of the class is created.
- In Python, the **.__init__()** method is called the **constructor** and is always called when an object is created.
- Note: Names that have lead and trailing double underscores are reserved for special use like the __init__ method for object constructors. These methods are known as **dunder methods**.

Syntax of Constructor Declaration

```
def __init__(self):
    # body of the constructor
```

Types of constructors

- **Default Constructor**: The default constructor is a simple constructor that doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed known as **self**.

- **Parameterized Constructor**: A constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as **self** and the rest of the arguments are provided by the programmer.
- The properties that all **Car** objects must have been defined in. **__init__()**. Every time a new **Car** object is created. **__init__()** sets the initial state of the object by assigning the values of the object's properties. That is **__init__()** initializes each new instance of the class.
- When a new class instance is created, the instance is automatically passed to the self parameter. **__init__()** so that new attributes can be defined on the object.

The self Parameter

- The **self** parameter is a reference to the current instance of the class and is used to access variables that belong to the class.
- It does not have to be named **self**, you can call it whatever you like, but it has to be the first parameter of any function in the class.
- You can give **.__init__()** any number of parameters, but the first parameter will always be a variable called self.

Let's update the Car class with the. **__init__()** method that creates n ame and **topSpeed** attributes:

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

Note: The **.__init__()** method's signature is indented in four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the **.__init__()** method belongs to the **Car** class.

In the body of **.__init__()**, two statements are using the self variable:

1. **self.name = name** creates an attribute called **name** and assigns to it the value of the **name** parameter.

2. **self.topSpeed= topSpeed** creates an attribute called t opSpeed and assigns to it the value of the **topSpeed** parameter.

Previous
Next
Notes

Instance Attributes

Attributes created in **.__init__()** are called **instance** attributes. An instance attribute's value is specific to a particular instance of the class. All **Car** objects have a **name** and a **topSpeed**, but the values for the **name** and **topSpeed** attributes will vary depending on the **Car** instance. Different objects of the **Car** class will have different names and top speeds.

Example:

```
class Student:
    def __init__(self,name,rollno):
        self.name = name
        self.rollno = rollno
        
    def printDetails(self):
        print("name: ",self.name)
        print("rollno: ",self.rollno)
        
        
s = Student("parikh",101)
s.printDetails()
```

output:

```
name:  parikh
rollno:  101
```

Class Attributes

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of**. __init__()**.

For example, the following **Car** class has a class attribute called **color** with the value **"Black"**:

```
class Car:
    # Class attribute
    color = "Black"


    def __init__(self, name, topSpeed):
          self.name = name
          self.topSpeed= topSpeed
```

- Class attributes are defined directly beneath the first line of the class name and are indented by four spaces.
- They must always be assigned an initial value.
- When an instance of the class is created, the class attributes are automatically created and assigned to their initial values.

You should use class attributes to define properties that should have the same value for every class instance and you must use instance attributes for properties that vary from one instance to another.
Now that we have a Car class, let’s create some cars!

Previous

Next

Introduction

Inheritance

● Inheritance is a powerful feature in Object-Oriented Programming.

● Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

● The class which inherits the properties of the other is known as **subclass** (derived class or child class) and the class whose properties are inherited is known as **superclass** (base class, parent class).

Let us take a real-life example to understand inheritance. Let's assume that Human is a class that has properties such as height, weight, age, etc and functionalities (or methods) such as eating(), sleeping(), dreaming(), working(), etc. Now we want to create Male and Female classes. Both males and females are humans and they share some common properties (like height, weight, age, etc) and behaviours (or functionalities like eating(), sleeping(), etc), so they can inherit these properties and functionalities from the Human class. Both males and females have some characteristics specific to them (like men have short hair and females have long hair). Such properties can be added to the Male and Female classes separately. This approach makes us write less code as both the classes inherit several properties and functions from the superclass, thus we didn't have to re-write them. Also, this makes it easier to read the code.

Python Inheritance Syntax

```python
class SuperClass:
  #Body of base class

class SubClass(BaseClass):
  #Body of derived class
```

The name of the superclass is passed as a parameter in the subclass while declaration.

Example:

To demonstrate the use of inheritance, let us take an example.
A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```python
class Polygon:
    def __init__(self, no_of_sides): #Constructor
    self.n = no_of_sides
    self.sides = [0 for i in range(no_of_sides)]

def inputSides(self): #Take user input for side lengths
    self.sides=[int(input("Enter side: "))for i in range(self.n)]


def displaySides(self): #Print the sides of the polygon
    for i in range(self.n):
    print("Side",i+1,"is",self.sides[i])
```

This class has **data attributes** to store the number of sides n and magnitude of each side as a list called **sides**.
The **inputSides()** method takes in the magnitude of each side and **dispSides()** displays these side lengths.

Now, a triangle is a **polygon** with 3 sides. So, we can create a class called **Triangle** which inherits from **Polygon**. In other words, we can say that every **triangle** is a **polygon**. This makes all the attributes of the **Polygon** class available to the **Triangle** class.

Constructor in Subclass

The constructor of the subclass must call the constructor of the superclass by accessing the **__init__** method of the superclass in the following format:

```
<SuperClassName>.__init__(self,<Parameter1>,<Parameter2>,...)
```

Note: The parameters being passed in this call must be the same as the parameters being passed in the superclass' **__init__** function, otherwise it will throw an error.

Syntax:

```
class subclass(superclass):
    def __init__(self):
        superclass.__init__(self,arg,...)
        #Calling constructor of the superclass
```

Example:

The **Triangle** class can be defined as follows.

```
class Polygon:

    #Constructor
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    #Take user input for side lenghts
    def inputSides(self):
        self.sides = [int(input("Enter Side: ")) for i in range(self.n)]

    #Print the sides of the polygon
    def displaySides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])

class Triangle(Polygon):

    def __init__(self):
        #Calling constructor of superclass
        Polygon.__init__(self, 3)

    def findArea(self):
        a, b, c = self.sides
        #Calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s * (s-a) * (s-b) * (s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, the class **Triangle** has a new method **findArea()** to find and print the area of the triangle. This method is only specific to the Triangle class and not the **Polygon** class. Here is a sample run:

output:

```
>>> t = Triangle() #Instantiating a Triangle object
>>> t.inputSides()
Enter side 1: 3
Enter side 2: 5
Enter side 3: 4


>>> t.dispSides()
Side 1 is 3
Side 2 is 5
Side 3 is 4
```

```
>>> t.findArea()
The area of the triangle is 6.00
```

We can see that even though we did not define methods like **inputSides()** or **displaySides()** for class Triangle separately, we were able to use them. If an attribute is not found in the subclass itself, the search continues to the superclass.

Please note that there are several types of inheritance in Python which are discussed in the upcoming topics. The one that we discussed in this section is the basic inheritance type and is called ***Single Inheritance.***

Previous

Next

Python super Keyword

Python super()

The super() built-in returns a proxy object (temporary object of the superclass) that allows us to access methods of the base class.

In Python, super() has two major use cases:

● Allows us to avoid using the base class name explicitly
● Working with Multiple Inheritance

Example:

In the case of single inheritance, it allows us to refer to the base class by super().

```
class Mammal(object):

  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):

  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog')

d1 = Dog()
```

output:

```
Dog has four legs.
Dog is a warm-blooded animal.
```

Here, we called the __init__() method of the Mammal class (from the Dog class) using code

```
super().__init__('Dog')
```

instead of

```
Mammal.__init__(self, 'Dog')
```

Previous

Next

Multilevel Inheritance

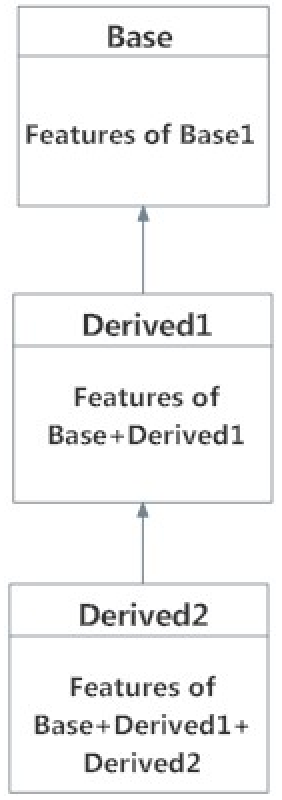
Multilevel Inheritance

We can also inherit from a derived class.
This is called **multilevel inheritance**. It can be of any depth in Python. An example is given below.

Syntax:

```
class SuperClass:
 pass

class SubClass(SuperClass):
 pass

class SubSubClass(SubClass):
 pass
```

Here, **SubClass** is derived from **SuperClass**, and **SubSubClass** is derived from **SubClass**.


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Example:

```
class A:

    def __init__(self,name):
        self.name = name

class B(A):

    def __init__(self,name,age):
        A.__init__(self,name)
        self.age = age

class C(B):
    def __init__(self,name,age,rollno):
        B.__init__(self,name,age)
        self.rollno = rollno


s = C("parikh",19,1011)
print(s.name)
print(s.age)
print(s.rollno)
```

output:

```
parikh
19
1011
```

Method Resolution Order (MRO)

Method Resolution Order(MRO) denotes the way a programming language resolves a method or an attribute. It defines the order in which the subclasses are searched when a method is called. First, the method or the attribute is searched within the subclass and then it is searched in its parent class and so on.

- Let's understand it through an example:

Example:

```
class Vehicle:

  def print(self):
      print("This is vehicle print")


class Car(Vehicle):

  def print(self):
      print("This is Car print")


class Waganor(Car):

   def print(self):
       print("This is Waganor print")


#Driver's code

w = Waganor()
w.print()
```

output:

```
This is Waganor print
```

In the above example, the method that is invoked is from the class ***Waganor*** but not from class ***Car*** or class ***Vehicle***, and this is due to the Method Resolution Order(MRO). The order of priority that is followed in the above code is- **class *Waganor* > class *Car* > class *Vehicle***.

if you want to get rid off of this ambiguity and want to call the print() method of any particular class we can use the syntax: **ClassName.method(object)**. Here binding the class ***Car*** and class ***vehicle*** with ***Waganor*** class object.

Example:

```
class Vehicle:

  def print(self):
      print("This is vehicle print")


class Car(Vehicle):

  def print(self):
      print("This is Car print")


class Waganor(Car):

   def print(self):
       print("This is Waganor print")


#Driver's code

w = Waganor()
Car.print(w)
Vehicle.print(w)
```

output:

```
This is Car print
This is vehicle print
```

Multiple Inheritance

● A class can be inherited from more than one superclass in Python, similar to C++. This is called multiple inheritances.
● In multiple inheritance, the features of all the superclasses are inherited into the subclass. The syntax for multiple inheritance is similar to single inheritance.

Syntax:

```
class SuperClass1:
 pass

class SuperClass2:
 pass

class SubClass(SuperClass1, SuperClass2):
 pass
```

Here, the SubClass class is derived from SuperClass1 and SuperClass2 classes and it has access to all the instance attributes and methods from both these superclasses.

Example:

```
class A:
    def test(self):
        print("print of A is called")

class B:
    def test(self):
        print("print of B is called")

class C(A,B):
    pass

#Driver's code

o = C()
o.test()
```

output:

```
print of A is called
```

In **multiple inheritances**, the methods are executed based on the **order specified while inheriting the classes** (Order inside parenthesis). Let’s look over another example to deeply understand the method resolution order:

```
class B:
  pass

class A:
  pass

class C(A, B):
  pass
```

The order in which the methods will be resolved will be C, A, B. This is because while inheriting the order is A, B.
Methods for Method Resolution Order(MRO) of a class:

To get the method resolution order of a class we can use the **mro()** method. By using this method we can display the order in which methods are resolved.

```
class B:
  pass

class A(B):
  pass


class C(A, B):
  pass


class D(C,A):
  pass

print(D.mro())
```

output:

```
[<class '__main__.D'>,<class '__main__.C'>,<class '__main__.A'>,<class '__main__.B'>,<class 'object'>]
```

Method Resolution Order (MRO)

Method Resolution Order(MRO) denotes the way a programming language resolves a method or an attribute. It defines the order in which the subclasses are searched when a method is called. First, the method or the attribute is searched within the subclass and then it is searched in its parent class and so on.

```
class Vehicle:

    def print(self):
        print("This is vehicle print")


class Car(Vehicle):

    def print(self):
        print("This is Car print")


#driver code
v = Car()
v.print()
```

output:

```
This is Car print
```

In the above example, the method that is invoked is from the class **Car** but not from class **Vehicle**, and this is due to the Method Resolution Order(MRO). The order of priority that is followed in the above code is- **class Car > class Vehicle**.

if you want to get rid off of this ambiguity and want to call the print() method of base class we can use syntax:- **ClassName.method(object)**. Here binding the class **Car** and class **Vehcile** with **Waganor** class object.

Example:

```
class Vehicle:

    def print(self):
        print("This is vehicle print")


class Car:
   def print(self):
       print("This is Car print")

class Waganor(Car, Vehicle):

    def print(self):
        print("This is Waganor print")



#Driver's code
w = Waganor()
w.print()
Car.print(w)
Vehicle.print(w)
```

output:

```
This is Waganor print
This is Car print
This is vehicle print
```

Previous

Next

Hierarchical Inheritance

Hierarchical Inheritance

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two children (derived) classes.

Syntax:

```
class Parent:
    pass

class child1(Parent):
    pass
```

```
class child2(Parent):
    pass
```

Example:

```
# Python program to demonstrate
# Hierarchical inheritance



# Base class
class Parent:
   def func1(self):
        print("This function is in the parent class.")


# Derived class1
class Child1(Parent):
   def func2(self):
        print("This function is in child 1.")


# Derivied class2
class Child2(Parent):
   def func3(self):
        print("This function is in child 2.")



#Driver's code

object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

output:

```
This function is in the parent class.
This function is in child 1.
This function is in the parent class.
This function is in child 2.
```

Previous

Next

Hybrid Inheritance

Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritances can be called hybrid inheritance.

Syntax:

```
class Parent:
    pass

class child1(Parent):
    pass

class child2(Parent):
    pass

class child3(child1,child2):
    pass
```

Example:

```
# Python program to demonstrate
# hybrid inheritance


class School:
def func1(self):
        print("This function is in school.")


class Student1(School):
```

```
def func2(self):
        print("This function is in student 1. ")


class Student2(School):
   def func3(self):
        print("This function is in student 2.")


class Student3(Student1, School):
   def func4(self):
        print("This function is in student 3.")


#Driver's code

object = Student3()
object.func1()
object.func2()
```

output:

```
This function is in school.
This function is in student 1.
```

Previous

Next

Notes

Python Static Method

Static methods, much like class methods, are methods that are bound to a class rather than its object. They do not require a class instance creation. So, they are not dependent on the state of the object. The difference between a static method and a class method is:

- The static method knows nothing about the class and just deals with the parameters.
- The class method works with the class since its parameter is always the class itself.

Hence, in newer versions of Python, you can use the @staticmethod decorator.

Syntax:

```
@staticmethod
def func(args,..):
    pass
```

Example:

```
class Hello:
    def __init__(self):
        pass
    
    @staticmethod
    def printHello():
        print("Hello World!")
        
o = Hello()
o.printHello()
```

output:

```
Hello World!
```

Python Instance Method

Instance methods are the most common type of methods in Python classes. These are so-called because they can access the unique data of their instance. If you have two objects each created from a car class, then they each may have different properties. They may have different colors, engine sizes, seats, and so on.

Instance methods must have self as a parameter, but you don't need to pass this in every time. Self is another Python special term. Inside any instance method, you can use self to access any data or methods that may reside in your class. You won't be able to access them without going through self.

Finally, as instance methods are the most common, there's no decorator needed. Any method you create will automatically be created as an instance method unless you tell Python otherwise.

Example:

```
class Student:
    name = “Parikh”
    
    #This is an instance method.
    def store_details(self):
        self.age = 60


    def print_age(self):
        print(self.age)

#Driver's code

s = Student()
s.store_details()
s.print_age()
```

output:

```
60
```

Previous
Next
Introduction

Access Modifiers

Access Modifiers are the specifications that can be defined to fix boundaries for classes when accessing member functions (methods) or member variables are considered. Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class. Most programming languages majorly have the following three forms of access modifiers, which are **Public**, **Private**, and **Protected** in a class.

Previous

Next

Public Modifiers

Public Modifier

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default. Consider the given example:

```
class Student:

 name = None # public member by default
 public age = None # public member
 # constructor
 def __init__(self, name, age):
     self.name = name
     self.age = age


#Driver's code

obj = Student("Boy", 15)
print(obj.age) #calling a public member of the class
print(obj.name) #calling a private member of the class
```

We will get the output as:

```
10
Boy
```

We will be able to access both **name** and **age** of the object from outside the class as they are **public**. However, this is not a good practice due to security concerns.

Previous

Next

Private Modifiers

Private Modifier

The members of a class that are declared **private** are accessible within the class only. A private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class. Consider the given example:

```
class Student:

 __name = None # private member
 age = None # public member

 def __init__(self, name, age): # constructor
     self.__name = name
     self.age = age
```

```
#Driver's code

obj = Student("Boy", 15)
print(obj.age) #calling a public member of the class
print(obj.name) #calling a private member of the class
```

We will get the output as:

```
10
AttributeError: 'Student' object has no attribute 'name'
```

We will get an **AttributeError** when we try to access the **name** attribute. This is because the **name** is a **private** attribute and hence it cannot be accessed from outside the class.
Note: We can even have **public** and **private** methods.

Private and Public modifiers with Inheritance

- The subclass will be able to access any public method or instance attribute of the superclass.
- The subclass will not be able to access any private method or instance attribute of the superclass.

Previous

Next

Protected Modifiers

Protected Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared **protected** by adding a single underscore '_' symbol before the data member of that class. The given example will help you get a better understanding:

```
# superclass
class Student:
  _name = None # protected data member

  # constructor
  def __init__(self, name):
      self._name = name
```

This is the parent class **Student** with a **protected** instance attribute **_name**. Now consider a subclass of this class:

```
class Student:
    __name = None # private member
    age = None # public member

    def __init__(self, name, age): # constructor
        self.__name = name
        self.age = age

#Driver's code

obj = Student("Boy", 15)
print(obj.age) #calling a public member of the class
print(obj.name) #calling a private member of the class

class Display(Student):
     # constructor
     def __init__(self, name):
        Student.__init__(self, name)

     def displayDetails(self):
        # accessing protected data members of the superclass
        print("Name: ", self._name)

#Driver's code
obj = Display("Boy") # creating objects of the derived class
obj.displayDetails() # calling public member functions of the class
obj.name # trying to access protected attribute
```

This class **Display** inherits the **Student** class. The method **displayDetails()** accesses the **protected** attribute **_name**. Further, we try to access it again outside this class.

output:

```
Name: Boy
AttributeError: 'Display' object has no attribute 'name'
```

You can observe that we were able to access the **protected** attribute **_name** from inside the **displayDetails()** method in the subclass. However, we were not able to access it outside the subclass and we got an **AttributeError**. This justifies the definition of the **protected** modifier.

Difference between private, protected, and public modifiers:

Let’s look at the summary of private, protected, and public access modifiers.

Visibility	Private	Protected	Public
Within the same class	Yes	Yes	Yes
In derived Class	No	Yes	Yes
Outside the class	No	No	Yes

Previous

Next

Polymorphism

Polymorphism

The literal meaning of polymorphism is the condition of occurrence in different forms. Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator, or object) to represent different types in different scenarios. Let's take a few examples:

Example 1: Polymorphism in addition(+) operator

We know that the **+** operator is used extensively in Python programs. But, it does not have a single usage. For integer data types, the **+** operator is used to perform an arithmetic addition operation.

```
num1 = 1
num2 = 2
print(num1+num2)
```

Hence, the above program outputs **3**.
Similarly, for string data types, the **+** operator is used to perform concatenation

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program outputs **"Python Programming"**.

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of **polymorphism** in Python.

Example 2: Functional Polymorphism in Python

There are some functions in Python which are compatible to run with multiple data types.
One such function is the len() function. It can run with many data types in Python. Let's look at some example use cases of the function

```
print(len("abcdefgeh"))
print(len(["a", "b", "c"]))
print(len({"a": 1, "b": 2}))
```

output:

```
9
3
2
```

Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the **len()** function. However, we can see that it returns specific information(the length) about the specific data types.

Class Polymorphism in Python

Polymorphism is a very important concept in Object-Oriented Programming. We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.
We can then later generalize calling these methods by disregarding the object we are working with.
Let's look at an example:

Example 3: Polymorphism in Class Methods

```
class Male:

 def __init__(self, name, age):
     self.name = name
     self.age = age
 def info(self):
     print("Hi, I am Male")

class Female:

 def __init__(self, name, age):
     self.name = name
     self.age = age
 def info(self):
     print("Hi, I am Female")
```

```
#Driver's code

M = Male("Sid", 20)
F = Female("Zee",21)
for human in (M, F): #Run loop over the set of objects
   human.info() #call the info function common to both
```

output:

```
Hi, I am Male
Hi, I am Female
```

Here, we have created two classes **Male** and **Female**. They share a similar structure and have the same method info(). However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through them using a common **human** variable. It is possible due to **polymorphism**. We can call both the **info()** methods by just using **human.info()** call, where human is first M (Instance of Male) and then F (Instance of **Female**).

Polymorphism and Inheritance

Like in other programming languages, the child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as **Method Overriding.**

Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class. Let's look at an example:

```
class Human:
 def __init__(self, name):
     self.name = name
 def info(self):
     print("Hi, I am Human”)


class Male(Human):

 def __init__(self, name):
     super().__init__(name)
 def info(self):
     print("Hi, I am Male”)
```

```
class Female(Human):

 def __init__(self, name):
     super().__init__(name)
 def info(self):
     print(“Hi, I am Female”)


#Driver's code

M = Male(“Sid”)
F = Female(“Zee”)
for human in (M, F): #Run loop over the set of objects
    human.info() #call the info function common to both
```

output:

```
Hi, I am Male
Hi, I am Female
```

Due to polymorphism, the Python interpreter automatically recognizes that the info() method for object M (**Male** class) is **overridden**. So, it uses the one defined in the subclass **Male**. Same with the object F (**Female** Class).
Note: Method Overloading, a way to create multiple methods with the same name but different arguments, is not possible in Python.

Previous

Next

Abstraction

Abstract Classes

Abstraction is the process of hiding implementation details and only revealing the parts of information which are relevant to the user i.e. it is basically, hiding the unnecessary details from the users. An abstract class can be considered as a blueprint for other classes that it can follow. Abstract classes are those type of classes that can contain one or more abstract methods. Here, an abstract method is a method that has a declaration but does not have an implementation (which is left for the inherited class to implement in its own form/fashion). This set of methods must be created within any child classes which inherit from the abstract class. A class that contains one or more abstract methods is called an **abstract class.**

Creating Abstract Classes in Python

● By default, Python does not provide abstract classes.

● Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is **abc**.

● abc works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base.

● A method becomes abstract when decorated with the keyword **@abstractmethod.**

● You are required to import **ABC** superclass and **abstractmethod** from the **abc** module before declaring your abstract class.

● An abstract class cannot be directly instantiated i.e. we cannot create an object of the abstract class.

● However, the subclasses of an abstract class that have definitions for all the abstract methods declared in the abstract class, can be instantiated.

● While declaring abstract methods in the class, it is not mandatory to use the **@abstractmethod** decorator (i.e it would not throw an exception). However, it is considered a good practice to use it as it notifies the compiler that the user has defined an abstract method.

The given Python code uses the **ABC** class and defines an abstract base class:

```
from abc import ABC, abstractmethod #Importing the ABC Module

class AbstractClass(ABC):
    def __init__(self, value): #Class Constructor
        self.value = value
        pass

    @abstractmethod
    def do_something(self): #Our abstract method declaration
        pass
```

● You are required to define (implement) all the abstract methods declared in an Abstract class, in all its subclasses to be able to instantiate the subclass.

For example, We will now define a subclass using the previously defined abstract class. You will notice that since we haven't implemented the **do_something** method, in this subclass, we will get an exception.

```
class TestClass(AbstractClass):
   pass #No definition for do_something method

x = TestClass(4)
```

We will get the output as:

```
TypeError: Can't instantiate abstract class TestClass with abstract methods do_something
```

We will do it the correct way in the following example, in which we define two classes inheriting from our abstract class:

```
class add(AbstractClass):
    def do_something(self):
        return self.value + 42


class mul(AbstractClass):
    def do_something(self):
        return self.value * 42


#Driver's code
x = add(10)
y = mul(10)
print(x.do_something())
print(y.do_something())
```

We get the output as:

```
52
420
```

Thus, we can observe that a class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Note: Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

● An abstract method can have an implementation in the abstract class.

● However, even if they are implemented, this implementation shall be overridden in the subclasses.

● If you wish to invoke the method definition from the abstract superclass, the abstract method can be invoked with super() call mechanism. (Similar to cases of “normal” inheritance).

● Similarly, we can even have concrete methods in the abstract class that can be invoked using super() call. Since these methods are not abstract it is not necessary to provide their implementation in the subclasses.

● Consider the given example:

Let's take an example of an abstract class *Payment*. It has an abstract method called, *paymentMode*. There are three classes derived from *Payment,* namely *CreditCardPayment, DebitCardPayment*, *and UPI_Payment.* All the three derived classes implement the abstract method *paymentMode* as one of their functionalities.

As a user, we are abstracted from the paymentMode that is inside the class ***Payment***. If we are creating another class ***CreditCardPayment*** that inherits the properties of class ***Payment*** then we must have to override all the abstract methods i.e paymentMethod here.

Example:

```
from abc import ABC, abstractmethod

class Payment(ABC):
    def printSlip(self, amount):
        print('Purchase of amount- ', amount)

    @abstractmethod
    def paymentMode(self, amount):
        pass

class CreditCardPayment(Payment):

    def paymentMode(self, amount):
        print('Credit card payment of- ', amount)

class DebitCardPayment(Payment):

    def paymentMode(self, amount):
        print('Debit card payment of- ', amount)

class UPI_Payment(Payment):

    def paymentMode(self,amount):
        print("UPI payment of- ",amount)



#Driver's code
object = CreditCardPayment()
object.paymentMode(100)
object.printSlip(100)

object = DebitCardPayment()
```

```python
object.paymentMode(200)
object.printSlip(200)

object = UPI_Payment()
object.paymentMode(300)
object.printSlip(300)
```

output:

```
Credit card payment of-  100
Purchase of amount-  100
Debit card payment of-  200
Purchase of amount-  200
UPI payment of-  300
Purchase of amount-  300
```

The given code shows another implementation of an abstract class.

```python
# Python program showing how an abstract class works
from abc import ABC, abstractmethod

class Animal(ABC): #Abstract Class
    @abstractmethod
    def move(self):
        pass


class Human(Animal): #Subclass 1
    def move(self):
        print("I can walk and run")


class Snake(Animal): #Subclass 2
    def move(self):
        print("I can crawl")


class Dog(Animal): #Subclass 3
    def move(self):
        print("I can bark")
```

```
# Driver's code
R = Human()
R.move()
K = Snake()
K.move()
R = Dog()
R.move()
```

We will get the output as:

```
I can walk and run
I can crawl
I can bark
```

Previous

Next

Introduction

Exception Handling

An Exception (error) is an event due to which the normal flow of the program's instructions gets disrupted.
Errors in Python can be of the following two types i.e. Syntax errors and Exceptions.

- While exceptions are raised when some internal events occur which changes the normal flow of the program.
- On the other hand, Errors are those type of problems in a program due to which the program will stop the execution.

Difference between Syntax Errors and Exceptions

Syntax Error: As the name that it has suggests that this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Example:

Consider the given code snippet:

```
val = 10

if(val > 20)
  print("Example")
```

We will get the output as:

Output:

```
SyntaxError: invalid syntax
```

The syntax error is because there should be a “:” (colon) at the end of an if statement. Since that is not present in the program, it gives a syntax error.

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

Consider the given code snippet:

```
balance = 10000
rem = balance / 0
print(rem)
```

We will get the output as:

Output:

```
ZeroDivisionError: division by zero
```

The above example raised the ZeroDivisionError exception, as we are trying to divide a number by 0 which is not defined and arithmetically not possible.

Exceptions in Python

- Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled.
- If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and the program comes to a sudden unexpected halt.

Some Common Exceptions

A list of common exceptions that can be thrown from a standard Python program is given below.

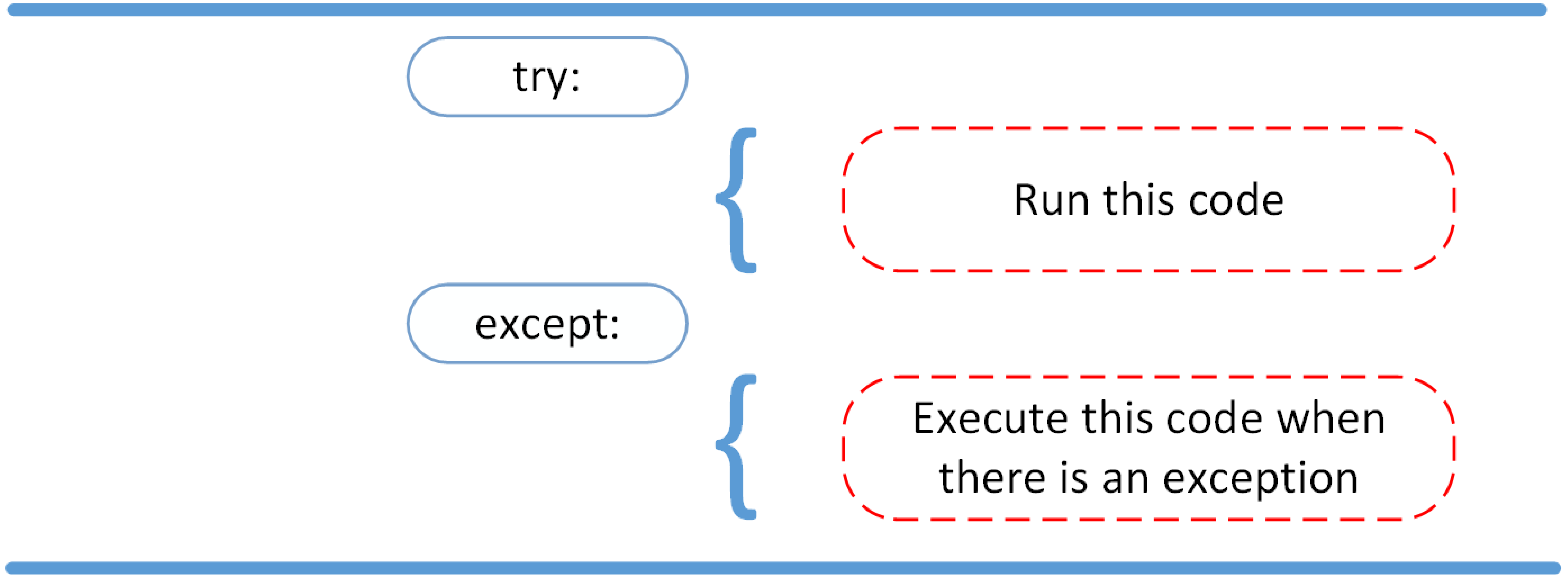
- **ZeroDivisionError:** This occurs when a number is divided by zero.
- **NameError:** It occurs when a name is not found. It may be local or global.
- **IndentationError:** It occurs when incorrect indentation is given.
- **IOError:** It occurs when an Input-Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

Previous
Next
Notes

Catching Exceptions

In Python, exceptions can be handled using try-except blocks.

- If the Python program contains suspicious code that may throw the exception, we must place that code in the try block.
- The try block must be followed by the except statement, which contains a block of code that will be executed in case there is some exception in the try block.
- We can thus choose what operations to perform once we have caught the exception.



Syntax:

```
try:
        # Some Code....

except:
        # optional block
        # Handling of exception (if required)
```

Example:

```
l = ['a', 0, 2]
```

```
for ele in l:
        try:
                print("The entry is", ele)
                r = 1/int(ele)

        except Exception as e: #Using Exception class
                print("Oops!", e.__class__, "occurred.")
                print("Next entry.")
                print()

        print("The reciprocal of", ele, "is", r)
```

We get the output to this code as:

```
The entry is a
Oops! <class 'ValueError'>occurred.

The entry is 0
Oops! <class 'ZeroDivisonError'>occured.

The entry is 2 T
The reciprocal of 2 is 0.5
```

- In this program, we loop through the values of a list l.
- As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the except block is skipped and normal flow continues(for last value).
- But if any exception occurs, it is caught by the except block (first and second values).
- Here, we print the name of the exception using the **exc_info()** function inside **sys** module.
- We can see that element “a” causes ValueError and 0 causes ZeroDivisionError.

Every exception in Python inherits from the base **Exception** class. Thus we can write the above code as:

```
l = ['a', 0, 2]
for ele in l:
 try:
     print("The entry is", ele)
     r = 1/int(ele)

 except Exception as e: #Using Exception class

```

```
     print("Oops!", e.__class__, "occurred.")
     print("Next entry.")
     print()
       
print("The reciprocal of", ele, "is", r)
```

Output:

```
This program has the same output as the above program.
```

Catching Specific Exceptions in Python

- In the above example, we did not mention any specific exception in the **except** clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
- We can specify which exceptions an **except** clause should catch.
- A try clause can have any number of **except** clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
- You can use multiple **except** blocks for different types of exceptions.
- We can even use a tuple of values to specify multiple exceptions in an **except** clause. Here is an example to understand this better:

Syntax:

```
try:
       # Some Code....
except:
       # optional block
       # Handling of exception (if required)
```

Example:

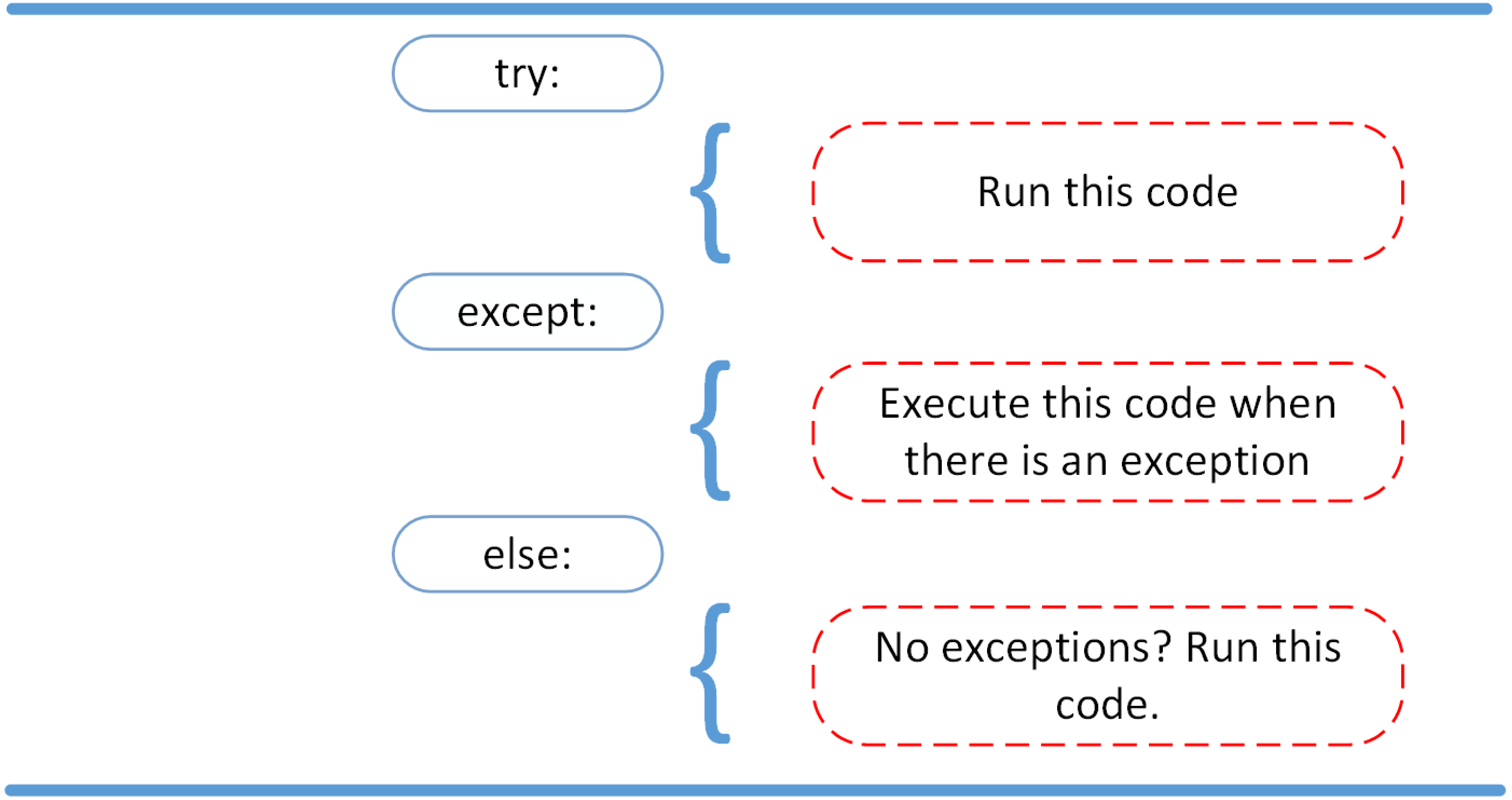
```
try:
    a=10/0
    
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
```

Output:

```
Arithmetic Exception
```

try-except-else Statements

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the else block. The syntax is given below:


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.

Syntax:

```
try:
        # Some Code....

except:
        # optional block
        # Handling of exception (if required)

else:
        # execute if no exception
```

Consider the example code to understand this better:

Example:

```
try:
    c = 2/1

except Exception as e:
    print("can't divide by zero")
    print(e)

else:
    print("Hi I am else block")
```

Output:

```
Hi I am else block
```

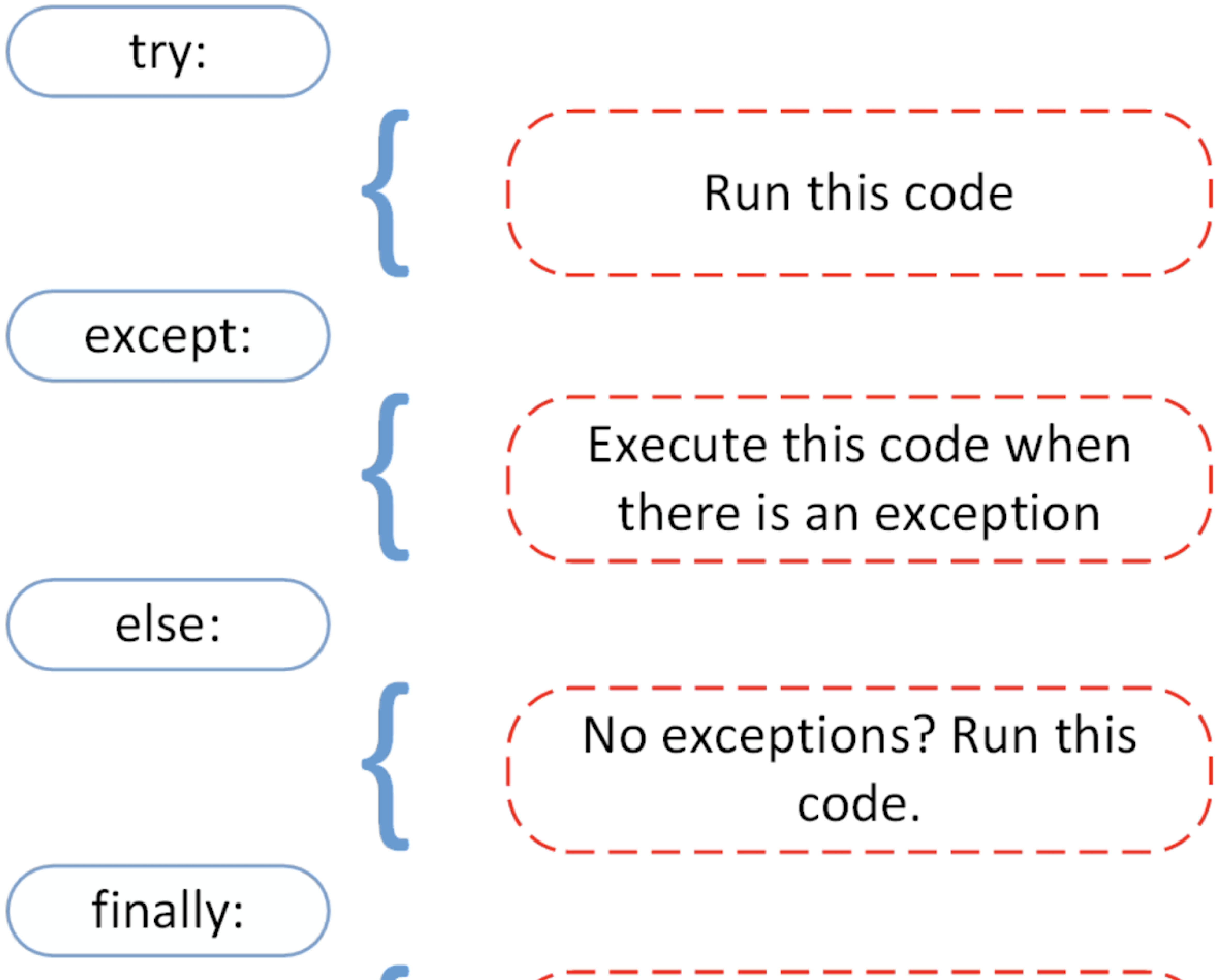
We get this output because there is no exception in the try block and hence the else block is executed. If there was an exception in the try block, the else block will be skipped and except block will be executed.

Previous

Next

Notes

finally Statement


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:

Syntax:

```
try:
       # Some Code....
       
except:
       # optional block
       # Handling of exception (if required)
       
else:
       # execute if no exception
       
finally:
      # Some code .....(always executed)
```

The **try** statement in Python can have an optional **finally** clause. This clause is executed no matter what and is generally used to release external resources. Here is an example of file operations to illustrate this:

Let's first understand how the try and except works –

- First, the try clause is executed i.e. the code between try and except clause.
- If there is no exception, then only the try clause will run, except the clause will not get executed.
- If any exception occurs, the try clause will be skipped and except clause will run.
- If any exception occurs, but the except clause within the code doesn't handle it, it is passed on to the outer try statements. If the exception is left unhandled, then the execution stops.
- A try statement can have more than one except clause.

Example: Let us try to take user integer input and throw the exception in except block.

```
# Python code to illustrate
# working of try()

def divide(x, y):

        try:
                # Floor Division : Gives only Fractional
                # Part as Answer
                result = x // y

        except ZeroDivisionError:
                print("Sorry ! You are dividing by zero ")
        else:
                print("Yeah ! Your answer is :", result)
        finally:
                # this block is always executed
                # regardless of exception generation.
                print('This is always executed')

# Look at parameters and note the working of Program
divide(3, 2)
divide(3, 0)
```

Output:

```
Yeah! Your answer is: 1
This is always executed
Sorry! You are dividing by zero
This is always executed
```

Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword. We can optionally pass values to the exception to clarify why that exception was raised. Given below are some examples to help you understand this better

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument
```

Now, consider the given code snippet:

Example:

```
try:
    a = -2
    if a <= 0:
        raise ValueError("That is not a positive number!")

except ValueError as ve:
    print(ve)
```

Output:

```
That is not a positive number!
```

Previous
Next
Introduction

Introduction

Collections is a built-in Python module that implements specialized container datatypes providing alternatives to Python’s general-purpose built-in containers such as ‘dict’, ‘list’, ‘set’, and ‘tuple’.
The collection Module in Python provides different types of containers which are objects that can be used to store different objects and provide a way to access the contained objects and iterate over them.

Some of the useful data structures present in this module are:

List of Module

- Counter
- OrderedDict
- DefaultDict
- ChainMap
- NamedTuple
- Deque
- UserDict
- UserList
- UserString

Previous
Next
DefaultDict

DefaultDict

A DefaultDict is also a sub-class to the dictionary and returns a new dictionary-like object. Whenever it is used to provide some default values for the key that does not exist it never raises a KeyError.

Syntax:

```
class collections.defaultdict(default_factory)
```

Here, default_factory is a function that provides the default value for the dictionary created. If this parameter is absent then there is a KeyError that gets raised otherwise the very first argument of 'defaultdict' is ‘default_factory’ as a default data type for the dictionary.

Initializing DefaultDict Objects

DefaultDict objects can be initialized using DefaultDict() method by passing the data type as an argument.

Example:

```
# Python program to demonstrate
# defaultdict
from collections import defaultdict

# Defining the dict
d = defaultdict(int)
# The default value is 0 so there is no need to enter the key first
print(d['B'])
# L = [1, 2, 3, 4, 2, 4, 1, 2]
L = ['A','B','C','D','E','A']
# Iterate through the list for keeping the count
for i in L:
    d[i] += 1

print(d)
```

Output:

```
0
defaultdict(<class 'int'>, {'E': 1, 'B': 1, 'A': 2, 'D': 1, 'C': 1})
```

Previous

Next

OrderedDict

OrderedDict

Ordered dictionaries are similar to regular dictionaries with some extra capabilities relating to ordering operations.
An OrderedDict is also a sub-class of dictionary but unlike a dictionary, it remembers the order in which the keys were inserted.
They have become less important now that the built-in ‘dict’ class gained the ability to remember insertion order (but this new behaviour has become guaranteed only after Python 3.7).

Syntax:

```
class collections.OrderDict()
```

Example:

```
# A Python program to demonstrate working
# of OrderedDict
from collections import OrderedDict

print("This is a standard python Dict:\n")
d = {}
d['John'] = 1
d['Mary'] = 2
d['Lucy'] = 3
d['Brian'] = 4
# The order of the keys and values inserted won't be retained here
for key, value in d.items():
        print(key, value)

print("\nThis is an Ordered Dict:\n")
od = OrderedDict()
od['John'] = 1
od['Mary'] = 2
od['Lucy'] = 3
od['Brian'] = 4
# The order of the keys and values inserted would be preserved
for key, value in od.items():
        print(key, value)
```

Output:

```
This is a standard python Dict:
```

```
Brian 4
Mary 2
John 1
Lucy 3

This is an Ordered Dict:

John 1
Mary 2
Lucy 3
Brian 4
```

Previous

Next

Counters

Counters

A counter is a subclass of the dictionary. It is a dict subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. It is used to keep the count of the elements in an iterable in the form of an unordered dictionary where the key represents the element in the iterable and value represents the count of that element in the iterable.

Note: It is equivalent to a bag or multiset of other languages.

Syntax:

```
class collections.Counter([iterable-or-mapping])
```

Initializing Counter Objects

A counter object can be initialized using the counter() function and this function can be called in one of the following ways:

- With a sequence of items
- With a dictionary containing keys and counts
- With keyword arguments mapping string names to counts

Example:

```
# A Python program to show different
# ways to create Counter
from collections import Counter

# With sequence of items
print(Counter('codingninjas'))

# with dictionary
print(Counter({'n': 3, 'i': 2, 'd': 1, 'o': 1, 'a': 1, 'g': 1, 'c': 1, 'j': 1, 's': 1}))

# with keyword arguments
print(Counter(n=3, i=2, d=1, o=1, a=1, g=1, c=1, j=1, s=1))
```

Output:

```
Counter({'n': 3, 'i': 2, 'o': 1, 'a': 1, 's': 1, 'd': 1, 'c': 1, 'j': 1, 'g': 1})
Counter({'n': 3, 'i': 2, 's': 1, 'o': 1, 'a': 1, 'd': 1, 'c': 1, 'j': 1, 'g': 1})
Counter({'n': 3, 'i': 2, 's': 1, 'o': 1, 'a': 1, 'd': 1, 'c': 1, 'j': 1, 'g': 1})
```

Previous

Next

ChainMap

ChainMap

A ‘ChainMap’ groups multiple dictionaries or other mappings together to create a single, updateable view. If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.
A ‘ChainMap’ class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls.
A ChainMap basically encapsulates many dictionaries into a single unit and returns a list of dictionaries.

Syntax:

```
class collections.ChainMap(dict1, dict2, dict3,...)
```

Example:

```
# Python program to demonstrate
# ChainMap


from collections import ChainMap
```

```python
# winnings of different

d1 = {'JS': 'ReactJS', 'TS': 'ReactTS'}
d2 = {'JS': 'NodeJS', 'TS': 'DenoJS'}
d3 = {'JS': 'AngularJS', 'TS': 'Angular'}
# Defining the chainmap
c = ChainMap(d1, d2, d3)

print(c)
```

Output:

```
ChainMap({'TS': 'ReactTS', 'JS': 'ReactJS'}, {'TS': 'DenoJS', 'JS': 'NodeJS'}, {'TS': 'Angular', 'JS': 'AngularJS'})
```

Accessing Keys and Values from ChainMap

Values from ChainMap can be accessed using the key name. They can also be accessed by using the keys() and values() method.

Example:

```python
# Python program to demonstrate
# ChainMap


from collections import ChainMap
# winnings of different

d1 = {'JS': 'ReactJS', 'TS': 'ReactTS'}
d2 = {'JS': 'NodeJS', 'TS': 'DenoJS'}
d3 = {'JS': 'AngularJS', 'TS': 'Angular'}
# Defining the chainmap
c = ChainMap(d1, d2, d3)

# Accessing Values using key name
print(c['TS'])

# Accesing values using values()
# method
print(c.values())

# Accessing keys using keys()
# method
print(c.keys())
```

Output:

```
ReactTS
ValuesView(ChainMap({'TS': 'ReactTS', 'JS': 'ReactJS'}, {'TS': 'DenoJS', 'JS': 'NodeJS'}, {'TS': 'Angular', 'JS': 'AngularJS'}
))
KeysView(ChainMap({'TS': 'ReactTS', 'JS': 'ReactJS'}, {'TS': 'DenoJS', 'JS': 'NodeJS'}, {'TS': 'Angular', 'JS': 'AngularJS'}))
```

Adding new dictionary
A new dictionary can be added by using the **new_child()** method. The newly added dictionary is added at the beginning of the ChainMap.

Example:

```
# printing chainMap
print ("All the ChainMap contents are : ")
print (chain)

# using new_child() to add new dictionary
chain1 = chain.new_child(dic3)

# printing chainMap
print ("Displaying new ChainMap : ")
print (chain1)
```

Output:

```
All the ChainMap contents are :
ChainMap({'TS': 'ReactTS', 'JS': 'ReactJS'}, {'TS': 'DenoJS', 'JS': 'NodeJS'}, {'TS': 'Angular', 'JS': 'AngularJS'})
Displaying new ChainMap :
ChainMap({'TS': 'VueTS', 'JS': 'VueJS'}, {'TS': 'ReactTS', 'JS': 'ReactJS'}, {'TS': 'DenoJS', 'JS': 'NodeJS'}, {'TS': 'Angular
', 'JS': 'AngularJS'})
```

Previous

Next

NamedTuple

NamedTuple

A NamedTuple is a function for tuples with **Named Fields** and can be seen as an extension of the built-in tuple data type. Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code and return a tuple object with names for each position which the ordinary tuples lack. For example, consider a tuple names of books where the first element represents the book's name, second represents the author's name and the third element represents the number of pages in the book. Suppose for calling book name instead of remembering the index position you can actually call the element by using the name argument, then it will be really easy for accessing tuples element. This functionality is provided by the NamedTuple.

Syntax:

```
class collections.namedtuple(typename, field_names)
```

Example:

```
# Python code to demonstrate namedtuple()

from collections import namedtuple

# Declaring namedtuple()
Book = namedtuple('Book',['name','author','pages'])

# Adding values
book = Book('Clean Code','Robert Cecil Martin','431')

# Access using index
print ("Getting the Book's author using index : ",end ="")
print (book[1])

# Access using name
print ("The Book's name using keyname : ",end ="")
print (book.name)
```

Output:

```
Getting the Book's author using index : Robert Cecil Martin
The Book's name using keyname : Clean Code
```

Conversion Operations
1. _make(): This function is used to return a namedtuple() from the iterable passed as argument.
2. _asdict(): This function returns the OrdereDict() as constructed from the mapped values of namedtuple().

Example:

```python
# Python code to demonstrate namedtuple()

from collections import namedtuple

# Declaring namedtuple()
Book = namedtuple('Book',['name','author','pages'])

# Adding values
book = Book('Clean Code','Robert Cecil Martin','431')

# Access using index
print ("Getting the Book's author using index : ",end ="")
print (book[1])

# Access using name
print ("The Book's name using keyname : ",end ="")
print (book.name)


# Adding values initializing iterable
li = ['Hatching Twitter', 'Nick Bilton', '320' ]

# initializing dict
di = { 'name' : "Atomic Habits", 'author' : 'James Clear' , 'pages' : '288' }

# using _make() to return namedtuple()
print ("The namedtuple instance using iterable is : ")
print (Book._make(li))

# using _asdict() to return an OrderedDict()
print ("The OrderedDict instance using namedtuple is : ")
print (book._asdict())
```

Output:

```
Getting the Book's author using index : Robert Cecil Martin
The Book's name using keyname : Clean Code
The namedtuple instance using iterable is :
Book(name='Hatching Twitter', author='Nick Bilton', pages='320')
The OrderedDict instance using namedtuple is :
OrderedDict([('name', 'Clean Code'), ('author', 'Robert Cecil Martin'), ('pages', '431')])
```

Deque

Deques are a generalization of basic stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction. Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.
Deque (Doubly Ended Queue) is the optimized list for quicker append and pop operations from both sides of the container. It provides O(1) time complexity for append and pop operations as compared to a list with O(n) time complexity.

Syntax:

```
class collections.deque(list)
```

Example:

```
# Python code to demonstrate working of append(), appendleft()

from collections import deque

# initializing deque
de = deque([10,9,8])

# using append() to insert element at right end inserts 'A' at the end of deque
de.append("A")

# printing modified deque
print ("The deque after appending at right is : ")
print (de)

# using appendleft() to insert element at right end inserts 6 at the beginning of deque
de.appendleft(6)

# printing modified deque
print ("The deque after appending at left is : ")
print (de)
```

Output:

```
The deque after appending at right is :
deque([10, 9, 8, 'A'])
The deque after appending at left is :
deque([6, 10, 9, 8, 'A'])
```

Removing Elements

Elements can also be removed from the deque from both ends. To remove elements from the right use the pop() method and to remove elements from the left use popleft() method.

Example:

```
# Python code to demonstrate working of pop(), and popleft()

from collections import deque

# initializing deque
python_libs = deque(['Django','Flask','Celery','NumPy','Pandas'])

# using pop() to delete element from right end deletes 'Pandas' from the right end of deque
python_libs.pop()

# printing modified deque
print ("The deque after deleting from right is : ")
print (python_libs)

# using popleft() to delete element from left end deletes 'Django' from the left end of deque
python_libs.popleft()

# printing modified deque
print ("The deque after deleting from left is : ")
print (python_libs)
```

Output:

```
The deque after deleting from right is :
deque(['Django', 'Flask', 'Celery', 'NumPy'])
The deque after deleting from left is :
```

```
deque(['Flask', 'Celery', 'NumPy'])
```

Previous

Next

UserDict

UserDict

This class simulates a dictionary. The instance's (contents) are kept in a regular dictionary, which is accessible via the data attribute of UserDict instances. If 'initialdata' is provided (which is the first parameter that a UserDict expects/takes), data is initialized with its contents; note that a reference to 'initialdata' will not be kept, allowing it to be used for other purposes.
UserDict is a dictionary-like container that acts as a wrapper around the dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from dict; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute. This container is used when someone wants to create their own dictionary with some modified or new functionality.

Syntax:

```
class collections.UserDict([initialdata])
```

Example:

```
# Python program to demonstrate userdict

from collections import UserDict

# Creating a Dictionary where deletion is not allowed
class MyDict(UserDict):

        # Function to stop deletion from dictionary
        def append(self, val):
            self.data.update(val)

        def __del__(self):
                raise RuntimeError("Cannot Delete elements")

        # Function to stop pop from dictionary
        def pop(self, s = None):
                raise RuntimeError("Cannot Delete elements")

        # Function to stop popitem from Dictionary
        def popitem(self, s = None):
                raise RuntimeError("Cannot Delete elements")

# Driver's code
d = MyDict({'One':1,'Two': 2,'Three': 3})

print(d.data)

d.append({'Four': 4})

print(d)

d.pop('One')
```

Output:

```
{'One': 1, 'Three': 3, 'Two': 2}
{'One': 1, 'Three': 3, 'Two': 2, 'Four': 4}
Traceback (most recent call last):
  File "main.py", line 47, in <module>
    d.pop('One')
  File "main.py", line 31, in pop
    raise RuntimeError("Cannot Delete elements")
RuntimeError: Cannot Delete elements
Exception ignored in: <bound method MyDict.__del__ of {'One': 1, 'Three': 3, 'Two': 2, 'Four': 4}>
Traceback (most recent call last):
  File "main.py", line 26, in __del__
RuntimeError: Cannot Delete elements
```

Previous

Next

UserList

UserList

The UserList class simulates a list in which the instance’s contents are kept in a regular list, which is accessible via the data attribute of UserList instances. The instance’s contents are initially set to a copy of the list, defaulting to the empty list []. The list can be iterable, for example, a real Python list or a UserList object.
UserList is a list-like container that acts as a wrapper around the list objects. This is useful when someone wants to create their own list with some modified or additional functionality.

Syntax:

```
class collections.UserList([list])
```

Example:

```
# Python program to demonstrate userlist

from collections import UserList

# Creating a List where deletion is not allowed
class MyList(UserList):

  # Function to stop deletion from List
  def remove(self, s = None):
      raise RuntimeError("Deletion is not allowed")

  # Function to stop pop from List
  def pop(self, s = None):
      raise RuntimeError("Deletion is not allowed")

# Driver's code
L = MyList(['A', 'B', 'C', 'D'])

print("Original List")

# Inserting to List o)
print(L)

# Deliting From List
L.remove()
d.pop('B')
```

Output:

```
Original List
After Insertion
['A', 'B', 'C', 'D', 'E']
Traceback (most recent call last):
  File "main.py", line 39, in <module>
    L.remove()
  File "main.py", line 22, in remove
```

```
    raise RuntimeError("Deletion is not allowed")
RuntimeError: Deletion is not allowed
```

Previous

Next

UserString

UserString

User string is a string-like container and just like UserDict and UserList, it acts as a wrapper around string objects. It is used when someone wants to create their own strings with some modified or additional functionality.

Syntax:

```
class collections.UserString(seq)
```

Example:

```
# Python program to demonstrate
# userstring


from collections import UserString


# Creating a Mutable String
class Mystring(UserString):

   # Function to append to
   # string
   def append(self, s):
         self.data += s

   # Function to rmeove from
   # string
   def remove(self, s):
         self.data = self.data.replace(s, "")


# Driver's code
```

```
s1 = Mystring("Coding")
print("Original String:", s1.data)

# Appending to string
s1.append("n")
print("String After Appending:", s1.data)

# Removing from string
s1.remove("g")
print("String after Removing:", s1.data)
```

Output:

```
Original String: Coding
String After Appending: Codingn
String after Removing: Codinn
```

Previous

Next

Notes

Python Shortcuts and Tricks

Example:

```
from collections import Counter

num_lst = [1, 1, 2, 3, 4, 5, 3, 2, 3, 4, 2, 1, 2, 3]

cnt = Counter(num_lst)
print(dict(cnt))

# first 2 most occurrence
print(dict(cnt.most_common(2)))
str_lst = ['blue', 'red', 'green', 'blue', 'red', 'red', 'green']

print(dict(Counter(str_lst)))
```

Output:

```
{1: 3, 2: 4, 3: 4, 4: 2, 5: 1}
{2: 4, 3: 4}
{'blue': 2, 'red': 3, 'green': 2}
```

for loop Minimized

Example:

```
lst = [1, 2, 3]
doubled = []

for num in lst:
    doubled.append(num*2)
print(doubled)

doubled = [num*2 for num in lst]
print(doubled)
```

Output:

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

Check All Chars Uppercase

Example:

```
import string

def is_upper(word):
    return all(c in string.ascii_uppercase for c in list(word))
    
print(is_upper('HUMANBEING'))
print(is_upper('humanbeing'))
```

Output:

```
True
```

```
False
```

for and if in One Line

Example:

```
d = [1, 2, 1, 3]
a = [each for each in d if each == 1]
print(a)
```

Output:

```
[1,1]
```

Loop With Index

Example:

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

for index, value in enumerate(lst):
   print(f"{index}, {value}")

for index, value in enumerate(lst, start=10):
   print(f"{index}, {value}")
```

Output:

```
0, a
1, b
2, c
3, d
10, a
11, b
12, c
13, d
```

Reverse a String or List

Example:

```
a = "humanbeing"
print("Reversed :",a[::-1])
b = [1, 2, 3, 4, 5]
print("Reversed :",b[::-1])
```

Output:

```
Reversed : gniebnamuh
Reversed : [5, 4, 3, 2, 1]
```

Join String and List Together

Example:

```
str1 = "do"
str2 = "more"
lst = ["Write", "less", "code"]
str3 = ' '.join(lst) + ', ' + str1 + ' ' + str2
print(str3)
```

Output:

```
Write less code, do more
```

Convert List to Comma Separated String

Example:

```
fruits = ['apple', 'mango', 'orange']
print(', '.join(fruits))

numbers = [1, 2, 3, 4, 5]
print(', '.join(map(str, numbers)))

items = [1, 'apple', 2, 3, 'orange']
print(', '.join(map(str, items)))
```

Output:

```
apple, mango, orange
1, 2, 3, 4, 5
1, apple, 2, 3, orange
```

Dictionary Get When Key Not Found

Example:

```python
d = {'a': 1, 'b': 2}
print(d.get('c'))
print(d.get('c', 3))
```

Output:

```
None
3
```

Sort Dictionary

Example:

```python
from operator import itemgetter

d = {'a': 10, 'b': 20, 'c': 5, 'd': 8, 'e': 5}

# sort by value
print(sorted(d.items(), key=lambda x: x[1]))

# sort by value
print(sorted(d.items(), key=itemgetter(1)))

# sort by key
print(sorted(d.items(), key=itemgetter(0)))

# sort by value and return keys
print(sorted(d, key=d.get))
```

Output:

```
[('c', 5), ('e', 5), ('d', 8), ('a', 10), ('b', 20)]
[('c', 5), ('e', 5), ('d', 8), ('a', 10), ('b', 20)]
[('a', 10), ('b', 20), ('c', 5), ('d', 8), ('e', 5)]
['c', 'e', 'd', 'a', 'b']
```

Swapping Values

Example:

```
a, b = 9, 10
a, b = b, a
print(a, b)
```

Output:

```
10, 9
```

Merge Dictionaries

Example:

```
d1 = {'a': 1}
d2 = {'b': 2}

print(dict(d1.items() | d2.items()))
print({**d1, **d2})

d1.update(d2)
print(d1)
```

Output:

```
{'a': 1, 'b': 2}
{'a': 1, 'b': 2}
{'a': 1, 'b': 2}
```

Easy Printing

Example:

```
row = [100, "android", "ios", "blackberry"]
print(', '.join(str(x) for x in row))
print(*row, sep=', ')
```

Output:

```
100, android, ios, blackberry
100, android, ios, blackberry
```

Previous

Next