Python Programming

Course Code : CSE3011
Course Credit : 3
Course Type : LP

Dr. Pranshu Pranjal

(MTech & PhD, IIT (ISM) Dhanbad)

Assistant Professor (AB 125/122)

Division - Artificial Intelligence & Machine Learning,

School of Computing Science and Engineering,

VIT Bhopal University, Bhopal-Indore Highway,

Kothrikalan, Sehore, Madhya Pradesh - 466114

Contact No.: {M} (+91) 8979609032

Email: pranshupranjal@vitbhopal.ac.in

Module 1

- A Brief History of Python, Different Versions, Python 2 vs Python 3
- Installing Python, Environment Variables
- Executing Python from the Command Line, Editing Python Files
- Basic Python Syntax (String Values, String Operators, Numeric Data)
- Types Conversions
- Simple Input and Output
- Language components - Control Flow structures and
- Syntax – Relational Operators - Logical Operators
- Bit Wise Operators
- Python for Windows

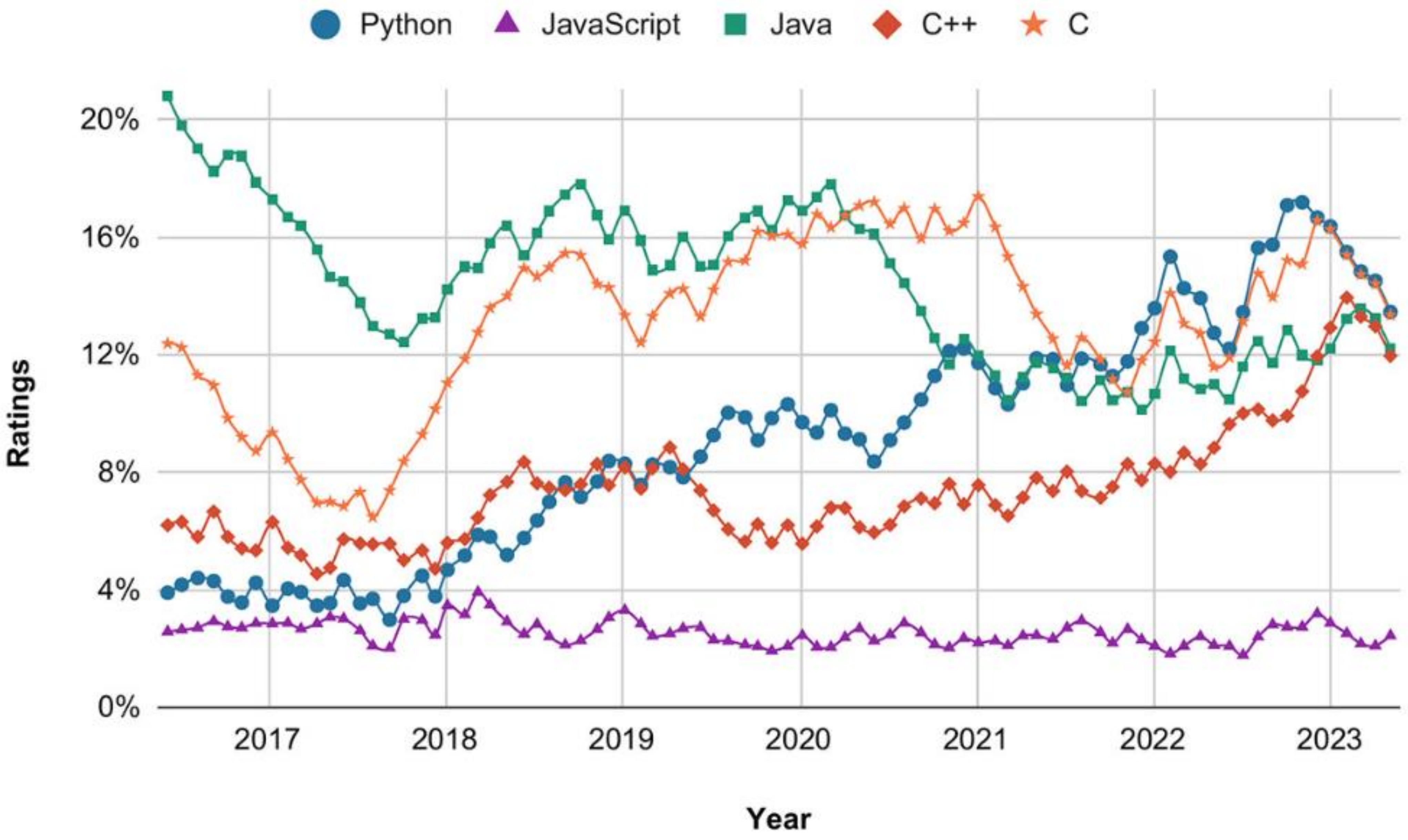
Historical background

Python has an interesting history. In 1982, Guido van Rossum, the creator of Python, started working at CWI, a Dutch national research institute. He joined a team that was designing a new programming language, named ABC, for teaching and prototyping. ABC's simplicity was ideal for beginners, but the language lacked features required to write advanced programs.

Several years later, van Rossum joined a different team at CWI working on an operating system. The team needed an easier way to write programs for monitoring computers and analyzing data. Languages common in the 1980's were (and still are) difficult to use for these kinds of programs. van Rossum envisioned a new language that would have a simple syntax, like ABC, but also provide advanced features that professionals would need.

At first, van Rossum started working on this new language as a hobby during his free time. He named the language Python because he was a fan of the British comedy group Monty Python. Over the next year, he and his colleagues successfully used Python many times for real work. van Rossum eventually decided to share Python with the broader programming community online. He freely shared Python's entire source code so that anyone could write and run Python programs.

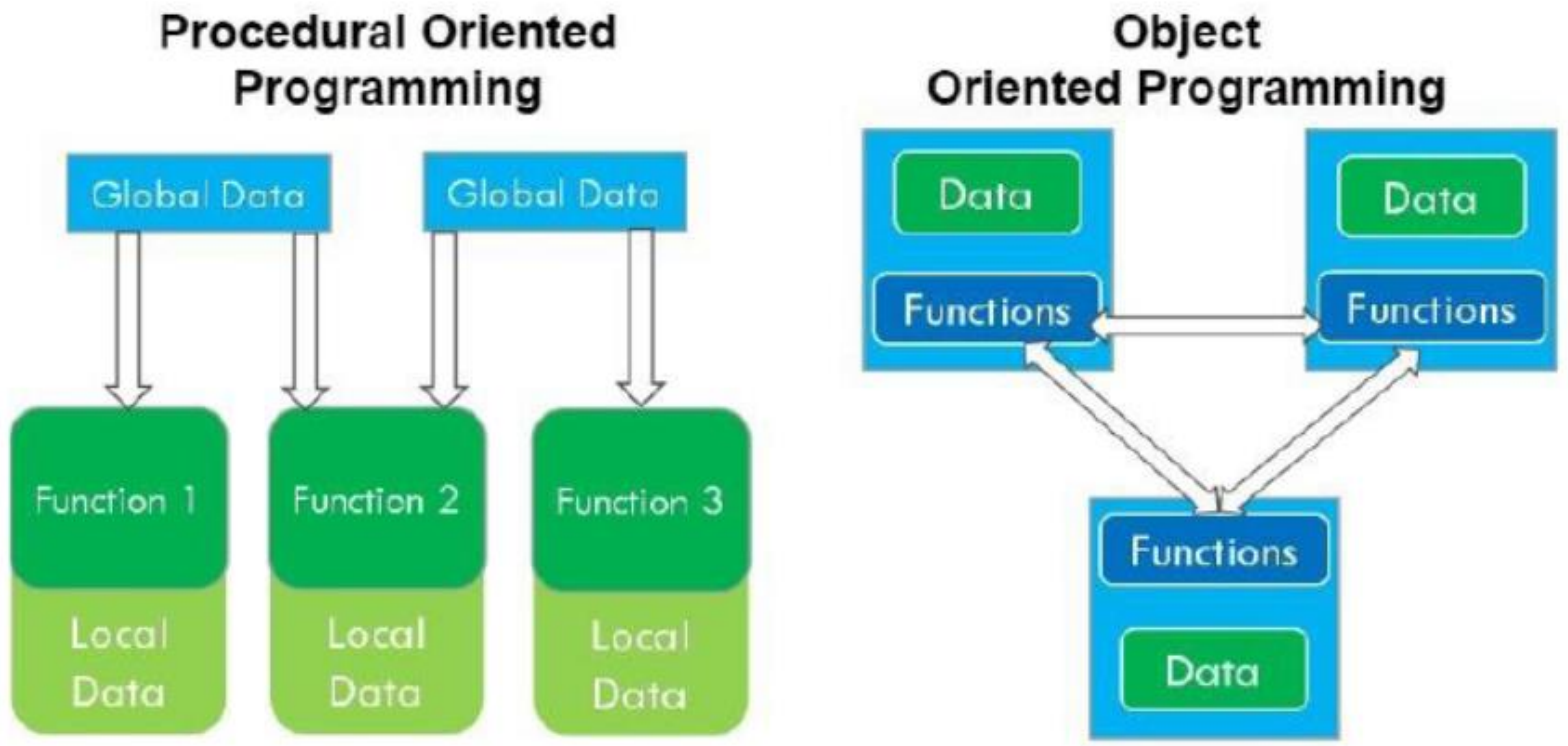
Python's first release, known as Version 0.9.0, appeared in 1991, about six years after C++ and four years before Java. van Rossum's decisions to make the language simple yet advanced, suitable for everyday tasks, and freely available online contributed to Python's long-term success.



Five of the most popular languages.

What is Python?

- Python is an open source, object-oriented, high-level powerful programming language.
- Developed by Guido van Rossumin the early 1989.
- Python runs on many Unix variants, on the Mac, and on Windows 2000 and later.



Prerequisites:

Knowledge of any programming language can be a plus.

Reason for increasing popularity

- Emphasis on code readability, shorter codes, ease of writing
- Programmers can express logical concepts in fewer lines of code in comparison to languages such as C++ or Java.
- Python supports multiple programming paradigms, like object-oriented, imperative and functional programming or procedural.
- There exists inbuilt functions for almost all of the frequently used concepts.
- Philosophy is “Simplicity is the best”.

LANGUAGE FEATURES

- **Interpreted**

There are no separate compilation and execution steps like C and C++.

- **Platform Independent**

Python programs can be developed and executed on multiple operating system platforms.

- **Free and Open Source;** Redistributable
- **High-level Language**

In Python, no need to take care about low-level details such as managing the memory used by the program.

- **Simple**

Closer to English language; Easy to Learn

- **Embeddable**

Python can be used within C/C++ program to give scripting capabilities for the program’s users.

- **Robust:**
- **Rich Library Support**

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.

Installing Python

Go to https://www.python.org, choose the Downloads menu, choose your operating system and a panel with a link to download the official package will appear:

Make sure you follow the specific instructions for your operating system. On macOS you can find a detailed guide on https://flaviocopes.com/pythoninstallation-macos/.

Brief Note on Python 2

- Python 2 is a version of the Python programming language that was released in 2000 and is now obsolete.
- It is replaced by Python 3, which includes many improvements over Python 2 and is not backward compatible.
- Python 2 has reached end-of-life status, meaning that it is no longer maintained or updated by the Python community.
- It is recommended to switch to Python 3 for all new development work.

Advantages of Python 2

- Easy to learn: Python 2 has a simple and straightforward syntax, making it easy for beginners to learn.
- Versatile: Python 2 can be used for a wide range of tasks, including web development, scientific computing, data analysis, artificial intelligence, and more.
- Large community: The Python community is vast, and there are many resources available to help you with any issues you may encounter while using Python 2.
- Large standard library: Python 2 has a large standard library that includes modules for many common programming tasks, saving you time and effort.
- Interoperable: Python 2 can be easily integrated with other languages and platforms, allowing for smooth and seamless collaboration.
- Dynamic typing: Python 2 is dynamically typed, meaning that variables do not have to be declared with a specific data type, making the code more flexible.
- Open-source: Python 2 is open-source software, which means that you can access the source code and make changes.

Disadvantages of Python 2

- No longer maintained: Python 2 has reached end-of-life status and is no longer maintained by the Python community, meaning that there are no security updates or bug fixes available.
- Incompatible with Python 3: Python 2 is not compatible with Python 3, so if you're starting a new project, you'll need to choose between the two.
- Lack of modern features: Python 2 is an older version of the language, and as a result, it lacks many of the features and improvements that have been added to Python 3.
- Performance limitations: Python 2 is slower than Python 3, and some tasks that would be quick and efficient in Python 3 may take much longer in Python 2.
- Limited support for Unicode: Python 2 has limited support for Unicode, which can cause issues when working with internationalized text.
- No support for type hints: Python 2 does not have support for type hints, which can make code maintenance and debugging more difficult.
- Deprecated packages and libraries: Many popular packages and libraries for Python 2 have been deprecated or no longer receive updates, so it can be more challenging to find up-to-date resources and support.

Brief Note on Python 3

- Python 3 is a high-level, interpreted programming language known for its simplicity and readability.
- It supports multiple programming paradigms including procedural, object-oriented, and functional programming.
- Python 3 has a large and active community, providing a vast library of modules for various applications including web development, scientific computing, and data analysis.
- It is widely used for scripting, automation, and developing standalone applications.

Advantages of Python 3

- Easy to Learn and Read: Python's syntax is simple and straightforward, making it easy to learn and understand, even for those with little or no programming experience.
- Large Community: Python has a large and active community, providing support, libraries, and tools for various applications.
- Versatile: Python can be used for a wide range of applications including web development, scientific computing, data analysis, artificial intelligence, and more.
- Cross-PlatformCompatibility: Python runs on various operating systems including Windows, Mac, and Linux, making it a platform-independent language.
- Rich Standard Library: Python has a vast and comprehensive standard library, providing a wide range of modules and tools for various applications.
- Dynamic Typing: Python supports dynamic typing, allowing variables to change their data type during runtime.
- Interoperability: Python can be integrated with other programming languages like C, C++, and Java, making it a flexible choice for complex applications.
- Large Ecosystem: Python has a large ecosystem of tools and libraries, including machine learning frameworks like TensorFlow, data visualization libraries like Matplotlib, and web frameworks like Django and Flask.

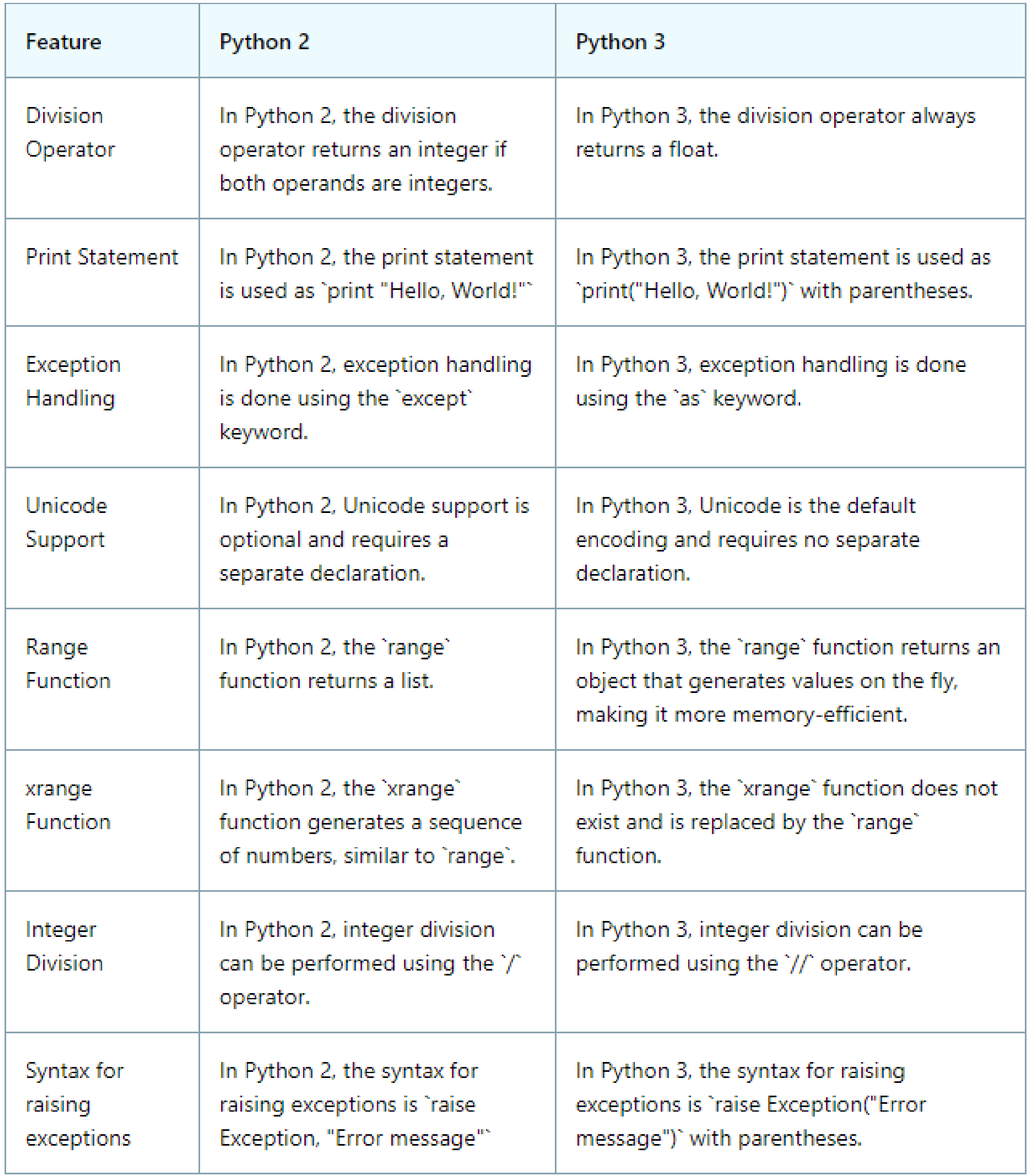
Disadvantages of Python 3

- Performance: Python can be slower than other programming languages like C or C++, especially for computational intensive tasks.
- Memory Management: Python uses a dynamic type system, which can lead to memory management issues, especially when working with large data sets.
- Not Suitable for Mobile Development: Python is not well suited for mobile app development and is mostly used for server-side or desktop applications.
- Global Interpreter Lock: Python uses a Global Interpreter Lock (GIL), which can limit the performance of multi-threaded applications and make it challenging to run multiple threads in parallel.
- Database Access: Python’s database access layer is weaker compared to other programming languages, and requires additional libraries or modules to interact with databases.
- Design Restrictions: Python is dynamically typed and does not enforce strict design restrictions, making it more prone to runtime errors and making it harder to catch bugs during development.
- Integration with Other Languages: Integrating Python with other programming languages can be challenging and require a deeper understanding of the underlying systems and libraries.
- Lack of Support for Low-Level Operations: Python is a high-level language and lacks support for low-level operations like pointer manipulation, making it challenging to develop system-level applications.

Key differences between Python 2 and 3

- Print statement: In Python 2, the print statement is used as "print" whereas in Python 3 it is used as "print()".
- Division: In Python 2, division of integers results in a floor division (rounded down to nearest whole number) while in Python 3, division results in a float.
- Strings: In Python 2, strings are ASCII encoded by default while in Python 3 they are Unicode encoded.
- Exception handling: In Python 2, "as" is used as an alias for catching exceptions while in Python 3 "as" is used as a keyword.
- Integer types: Python 3 removed the distinction between long integers and regular integers, and they are now simply referred to as integers.
- Range function: In Python 2, range() generates a list, whereas in Python 3 it generates an object of type range, which is iterable.
- Metaclasses: In Python 2, you need to use the "metaclass" argument when defining a class, whereas in Python 3 this is done using the "metaclass" keyword argument in class definition.
- Raise statement: In Python 2, the "raise" statement is used to raise an exception, whereas in Python 3 it is used as an expression to raise exceptions.
- Input method: In Python 2, the "input" method takes input as a string, while in Python 3 it takes it as an expression.

Python 2 and 3 are two versions of the Python programming language, with Python 3 being the latest and containing new features, syntax and improved performance. Python 2 has reached end-of-life and is no longer updated, while Python 3 is actively maintained. It is recommended to use Python 3 for new projects.



Feature	Python 2	Python 3
Division Operator	In Python 2, the division operator returns an integer if both operands are integers.	In Python 3, the division operator always returns a float.
Print Statement	In Python 2, the print statement is used as `print "Hello, World!"`	In Python 3, the print statement is used as `print("Hello, World!")` with parentheses.
Exception Handling	In Python 2, exception handling is done using the `except` keyword.	In Python 3, exception handling is done using the `as` keyword.
Unicode Support	In Python 2, Unicode support is optional and requires a separate declaration.	In Python 3, Unicode is the default encoding and requires no separate declaration.
Range Function	In Python 2, the `range` function returns a list.	In Python 3, the `range` function returns an object that generates values on the fly, making it more memory-efficient.
xrange Function	In Python 2, the `xrange` function generates a sequence of numbers, similar to `range`.	In Python 3, the `xrange` function does not exist and is replaced by the `range` function.
Integer Division	In Python 2, integer division can be performed using the `/` operator.	In Python 3, integer division can be performed using the `//` operator.
Syntax for raising exceptions	In Python 2, the syntax for raising exceptions is `raise Exception, "Error message"`	In Python 3, the syntax for raising exceptions is `raise Exception("Error message")` with parentheses.

Similarities between Python 2 and 3

- Syntax: The basic syntax and structure of Python 2 and Python 3 are almost the same.
- Variables and data types: Both Python 2 and Python 3 have the same type of variables and data types.
- Indentation: Indentation is still used in both versions to define blocks of code.
- Standard Library: Both versions have a large standard library with modules for various tasks, such as file I/O, regular expressions, and networking.
- Interactive Mode: Both versions have an interactive mode where you can enter code and see the results immediately.
- Object-Oriented Programming: Both versions support Object-Oriented Programming.
- Dynamic Typing: Both versions have dynamic typing, where variables don't have to be declared with a specific data type.
- Community: Both versions have a large and active community of developers who contribute to the development of the language and provide support.

Access environment variable values in Python

- An environment variable is a variable that is created by the Operating System.
- Environment variables are created in the form of Key-Value pairs.
- To Access environment variables in Python's we can use the OS module which provides a property called environ that contains environment variables in key-value pairs. We will see how to use environment variables in Python.

How to Use Environment Variables in Python?

These are the different ways of accessing environment variables: Using; os.environ()

- Access All Environment Variables
- Access Single Environment Variable
- Get value of the Environment variable
- Using os.getenv()
- Using python-dotenv Package

Access All Environment Variables Using os.environ()

```
# import os module
import os

# display all environment variable
print(os.environ)
```

Access Single Environment Variable Using os.environ()

```
# import os module
import os

# access environment variable
print(os.environ['COMPUTERNAME'])
```

Output

```
DESKTOP-M2ASD91
```

Get value of the environment variable key using os.environ

```
# import os module
import os

# access environment variable using the key
print(os.environ.get('USERPROFILE'))
```

Output

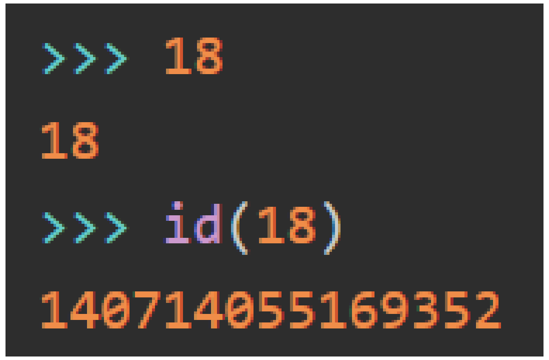
```
C:\Users\pranshu
```

Jupyter - Installation and Setup

- To launch and setup Jupyter, we will install Anaconda.
- Anaconda is the World's Most Popular Data Science Platform. It is useful for data science, machine learning applications, predictive analytics, etc.

Python Variables

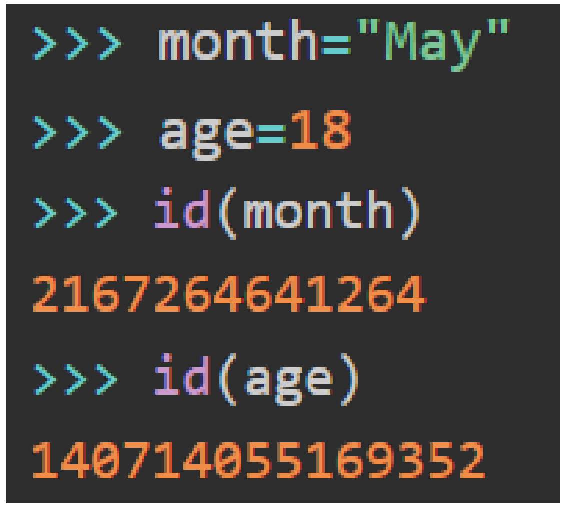
- Python variables are the reserved memory locations used to store values with in a Python Program. This means that when you create a variable you reserve some space in the memory.
- Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.
- It stores the object in a randomly chosen memory location. Python's built-in **id()** function returns the address where the object is stored.



```
>>> 18
18
>>> id(18)
140714055169352
```

Once the data is stored in the memory, it should be accessed repeatedly for performing a certain process. Obviously, fetching the data from its ID is cumbersome. High level languages like Python make it possible to give a suitable alias or a label to refer to the memory location.

Let us label the location of May as month, and location in which 18 is stored as age. Python uses the assignment operator (=) to bind an object with the label.

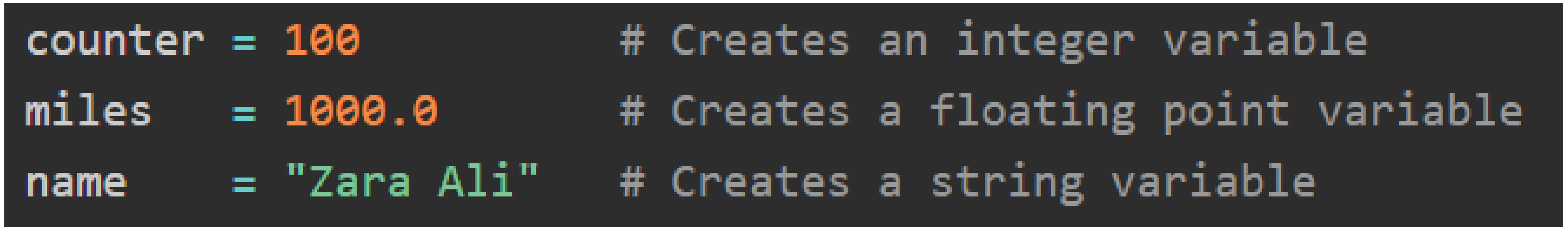


```
>>> month="May"
>>> age=18
>>> id(month)
2167264641264
>>> id(age)
140714055169352
```

Creating Python Variables

Python variables do not need explicit declaration to reserve memory space or you can say to create a variable. A Python variable is created automatically when you assign a value to it. The equal sign (=) is used to assign values to variables.

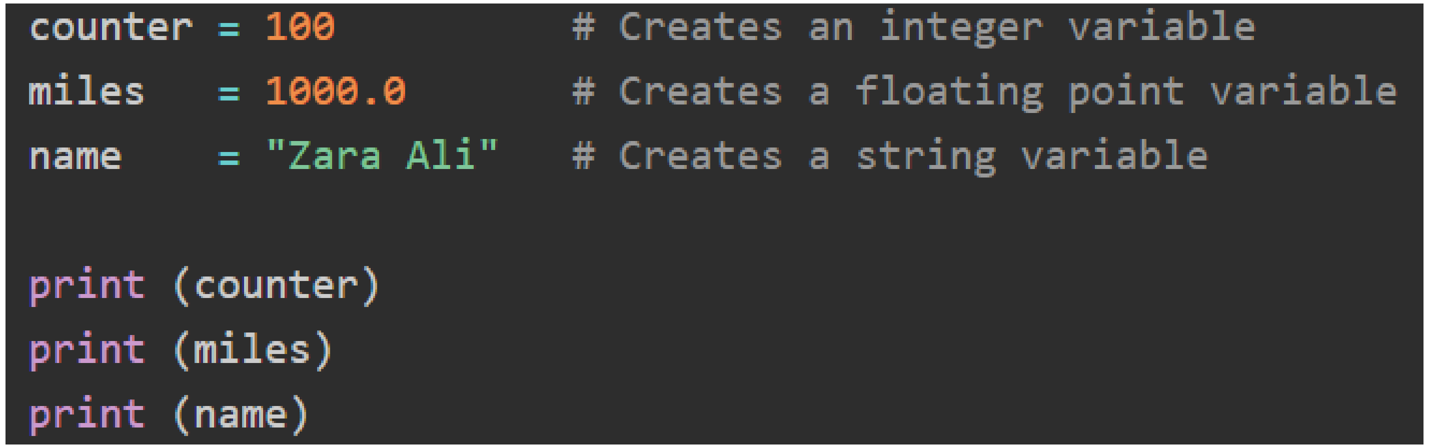
Example: This example creates different types (an integer, a float, and a string) of variables.



```
counter = 100          # Creates an integer variable
miles   = 1000.0       # Creates a floating point variable
name    = "Zara Ali"   # Creates a string variable
```

Printing Python Variables

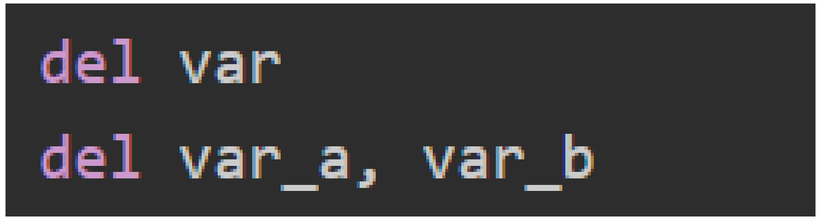
Once we create a Python variable and assign a value to it, we can print it using **print()** function. Following is the extension of previous example and shows how to print different variables in Python:



```
counter = 100          # Creates an integer variable
miles   = 1000.0       # Creates a floating point variable
name    = "Zara Ali"   # Creates a string variable

print (counter)
print (miles)
print (name)
```

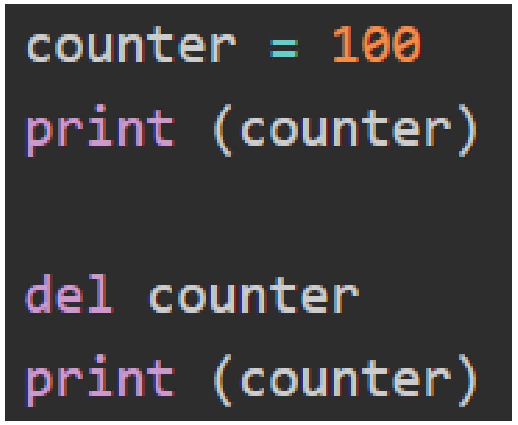
Deleting Python Variables

You can delete a single object or multiple objects by using the **del** statement. For example –



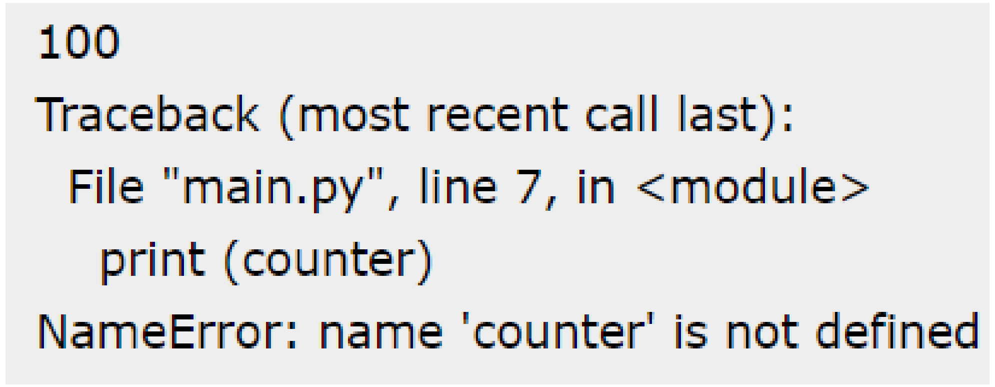
```
del var
del var_a, var_b
```

Following examples shows how we can delete a variable and if we try to use a deleted variable then Python interpreter will throw an error:



```
counter = 100
print (counter)

del counter
print (counter)
```

Result:

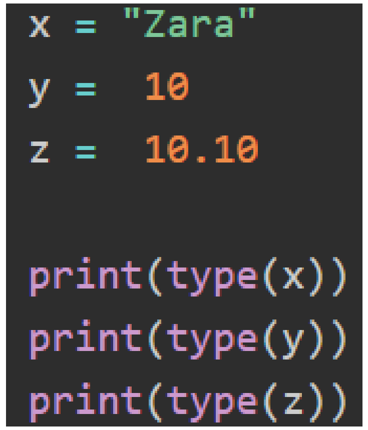


```
100
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print (counter)
NameError: name 'counter' is not defined
```

Getting Type of a Variable

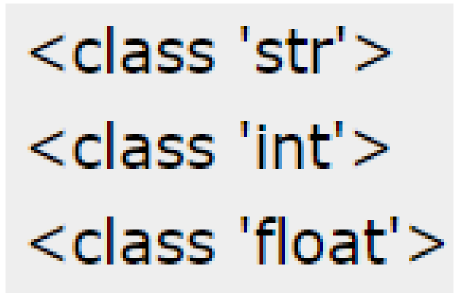
You can get the data type of a Python variable using the python built-in function **type()** as follows.

Example: Printing Variables Type



```
x = "Zara"
y =  10
z =  10.10

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

Result:

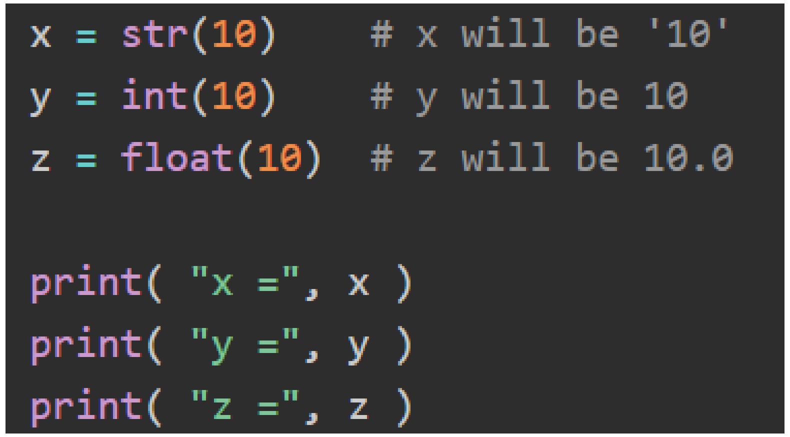


```
<class 'str'>
<class 'int'>
<class 'float'>
```

Casting Python Variables

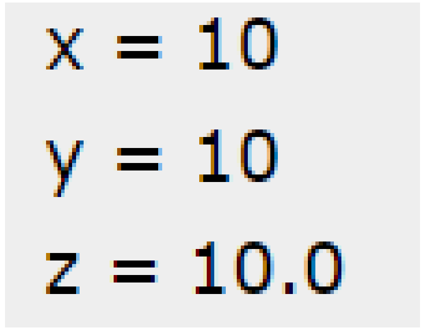
You can specify the data type of a variable with the help of casting as follows:

Example: This example demonstrates case sensitivity of variables.



```
x = str(10)     # x will be '10'
y = int(10)     # y will be 10
z = float(10)   # z will be 10.0

print( "x =", x )
print( "y =", y )
print( "z =", z )
```

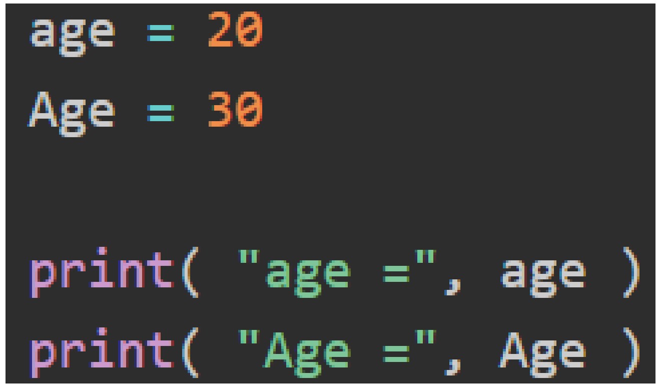
Result:



```
x = 10
y = 10
z = 10.0
```

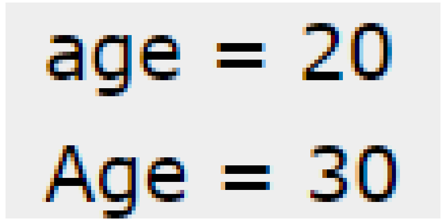
Case-Sensitivity of Python Variables

Python variables are case sensitive which means Age and age are two different variables:



```
age = 20
Age = 30

print( "age =", age )
print( "Age =", Age )
```

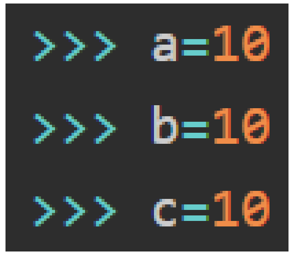
Result:



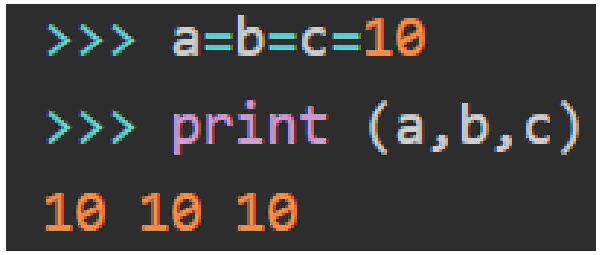
```
age = 20
Age = 30
```

Python Variables - Multiple Assignment

Python allows to initialize more than one variables in a single statement. In the following case, three variables have same value.



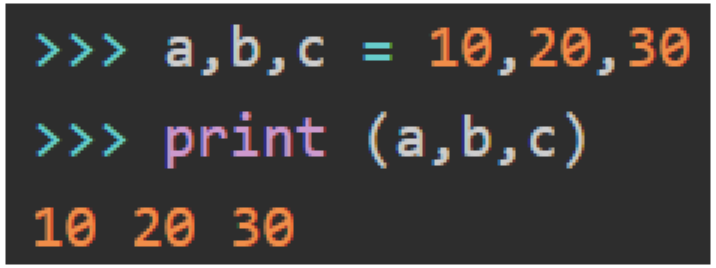
```
>>> a=10
>>> b=10
>>> c=10
```



```
>>> a=b=c=10
>>> print (a,b,c)
10 10 10
```

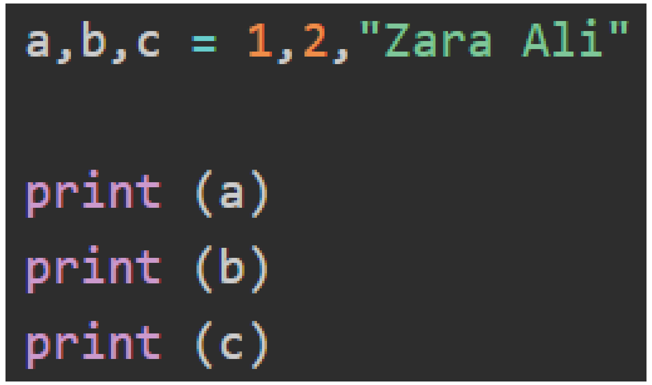
Instead of separate assignments, you can do it in a single assignment statement as

These separate assignment statements can be combined in one. You need to give comma separated variable names on left, and comma separated values on the right of = operator.



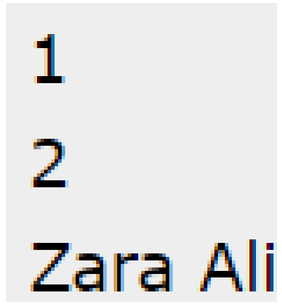
```
>>> a,b,c = 10,20,30
>>> print (a,b,c)
10 20 30
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Zara Ali" is assigned to the variable c.



```
a,b,c = 1,2,"Zara Ali"

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

Result:



```
1
2
Zara Ali
```

Python Variables - Naming Convention

Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like color, age, name etc. There are certain rules which should be taken care while naming a Python variable:

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

If the name of variable contains multiple words, we should use these naming patterns −

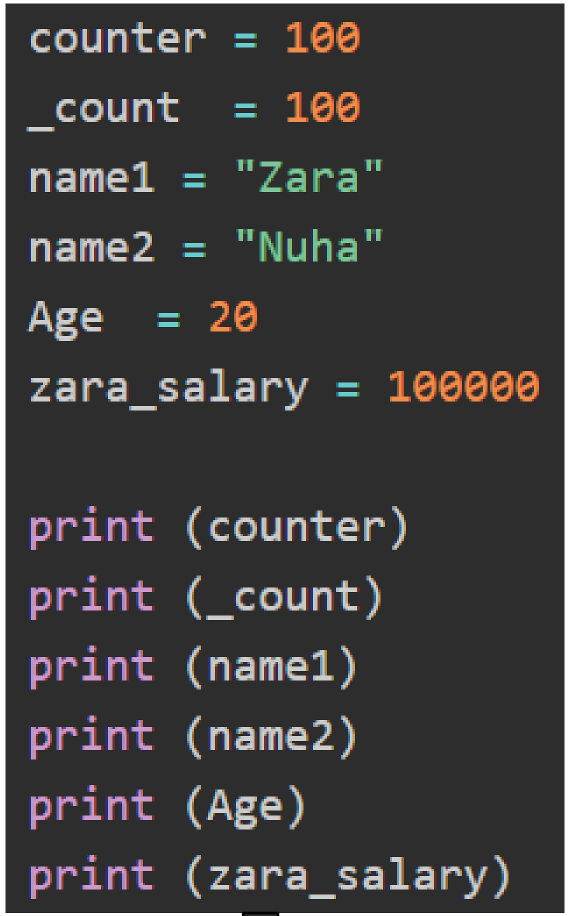
- **Camel case −** First letter is a lowercase, but first letter of each subsequent word is in uppercase. For example: kmPerHour, pricePerLitre
- **Pascal case −** First letter of each word is in uppercase. For example: KmPerHour, PricePerLitre
- **Snake case −** Use single underscore (_) character to separate words. For example: km_per_hour, price_per_litre

Python has reserved words, known as **keywords**, which have special functions and cannot be used as names for variables (or other objects).

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

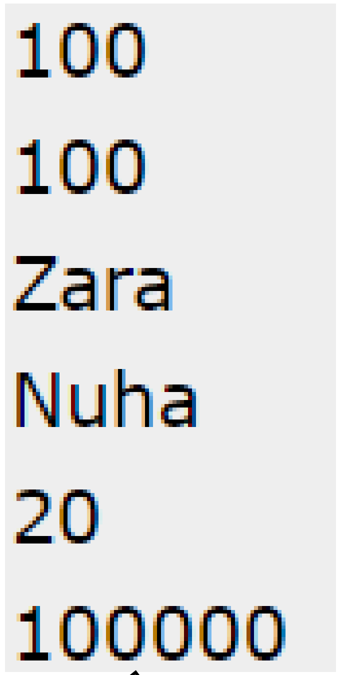
Example

Following are valid Python variable names:

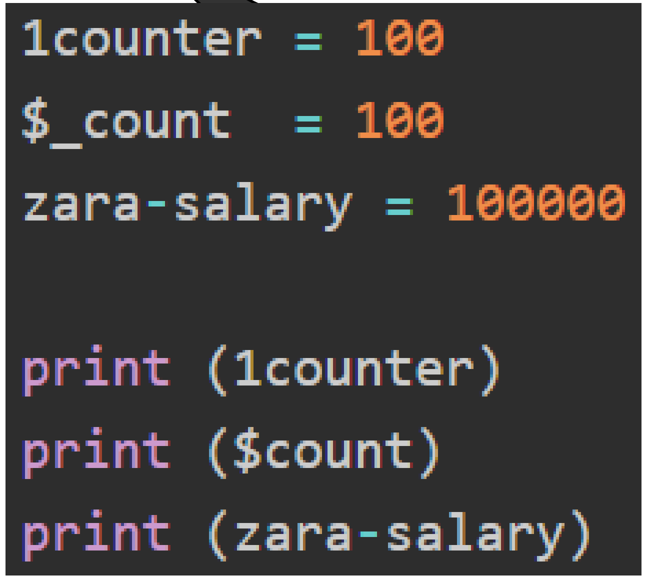


```python
counter = 100
_count  = 100
name1 = "Zara"
name2 = "Nuha"
Age   = 20
zara_salary = 100000

print (counter)
print (_count)
print (name1)
print (name2)
print (Age)
print (zara_salary)
```

Result:

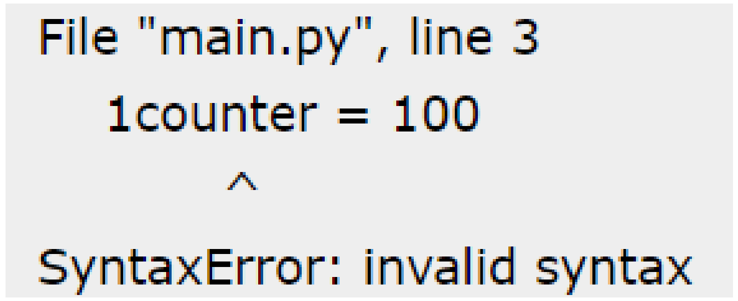


```
100
100
Zara
Nuha
20
100000
```



```python
1counter = 100
$_count  = 100
zara-salary = 100000

print (1counter)
print ($count)
print (zara-salary)
```

Result:



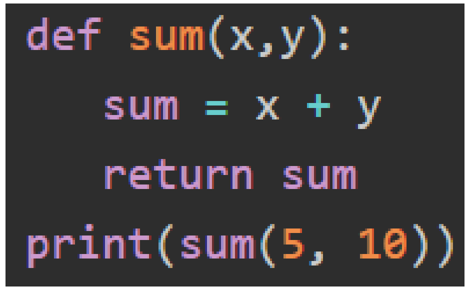
```
File "main.py", line 3
    1counter = 100
         ^
SyntaxError: invalid syntax
```

Python Local Variables

Python Local Variables are defined inside a function. We can not access variable outside the function.

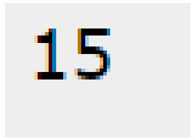
Example

Following is an example to show the usage of local variables:



```python
def sum(x,y):
    sum = x + y
    return sum
print(sum(5, 10))
```

Result:

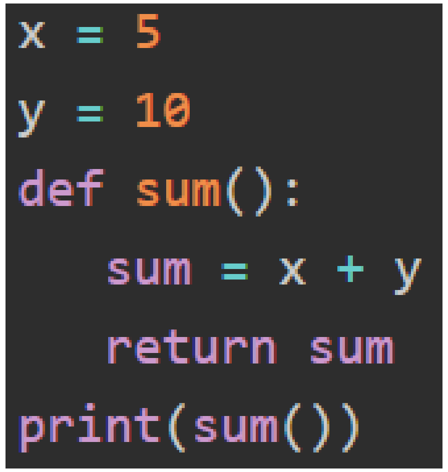


```
15
```

Python Global Variables

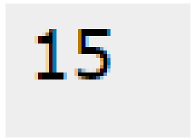
Any variable created outside a function can be accessed within any function and so they have global scope.

Example
Following is an example of global variables –



```python
x = 5
y = 10
def sum():
    sum = x + y
    return sum
print(sum())
```

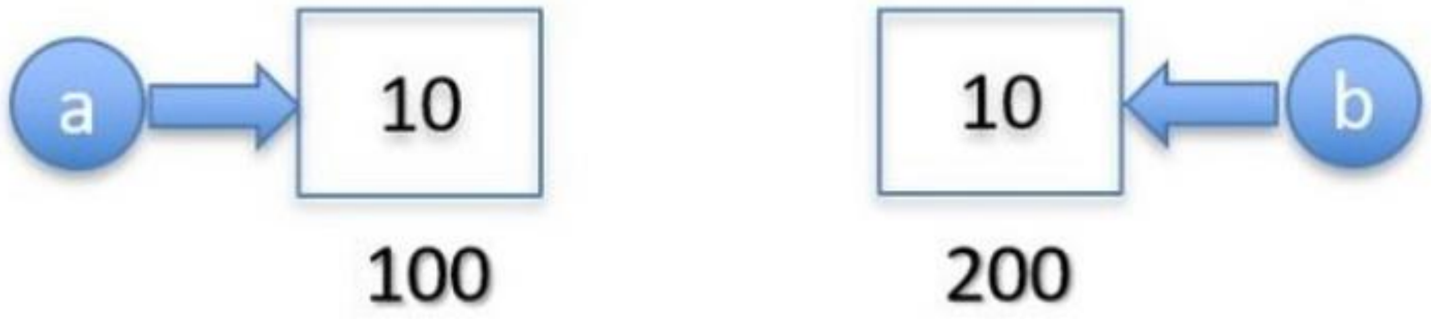
Result:



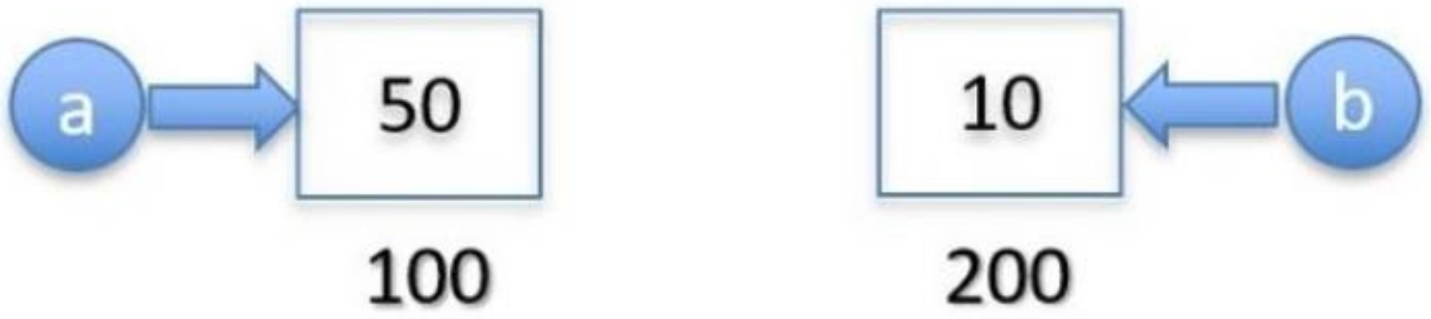
```
15
```

Python vs C/C++ Variables

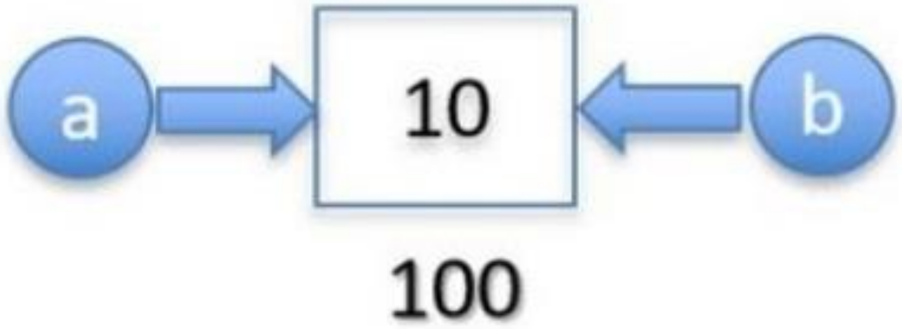
The concept of variable works differently in Python than in C/C++. In C/C++, a variable is a named memory location. If a=10 and also b=10, both are two different memory locations. Let us assume their memory address is 100 and 200 respectively.



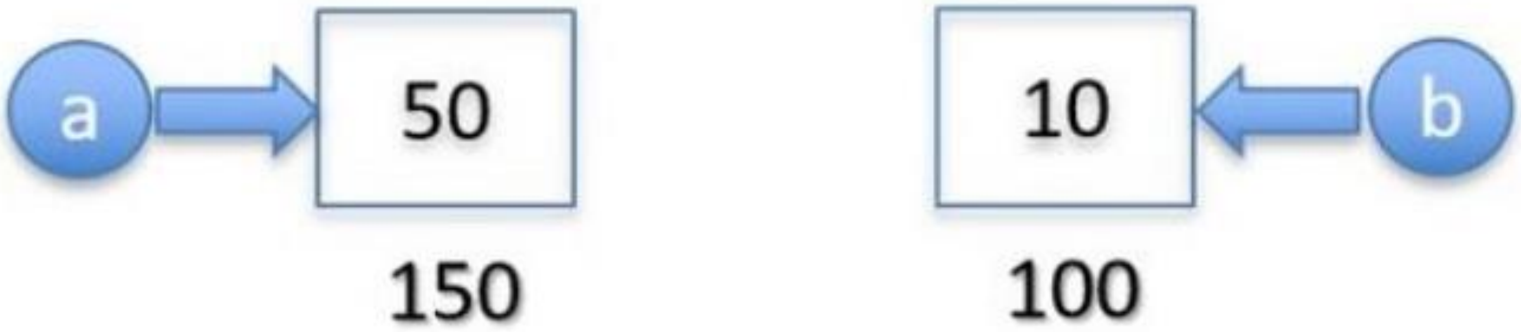
If a different value is assigned to "a" - say 50, 10 in the address 100 is overwritten.



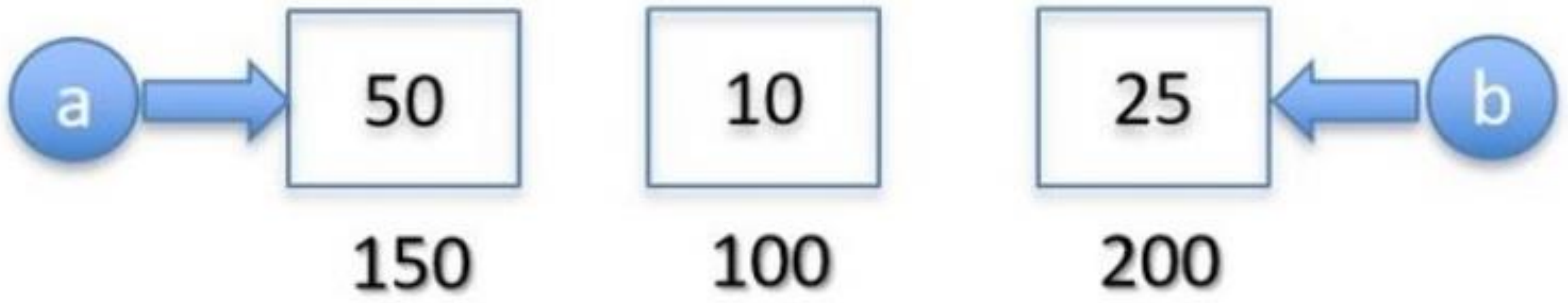
A Python variable refers to the object and not the memory location. An object is stored in memory only once. Multiple variables are really the multiple labels to the same object.



The statement a=50 creates a new **int** object 50 in the memory at some other location, leaving the object 10 referred by "b".



Further, if you assign some other value to b, the object 10 remains unreferred.



Python's garbage collector mechanism releases the memory occupied by any unreferred object.

Python's identity operator **is** returns True if both the operands have same id() value.

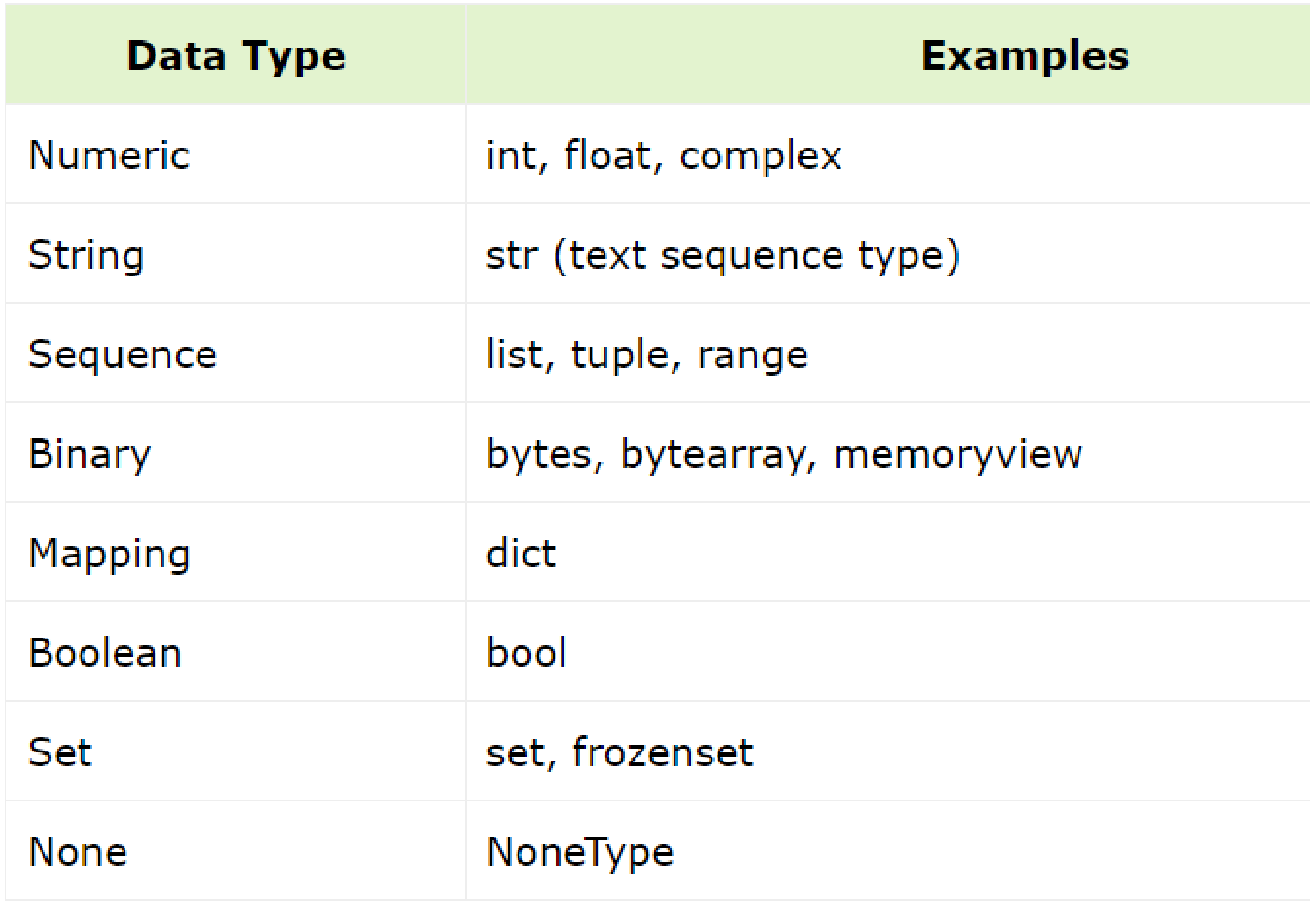
```
>>> a=b=10
>>> a is b
True
>>> id(a), id(b)
(140731955278920, 140731955278920)
```

Data Types in Python

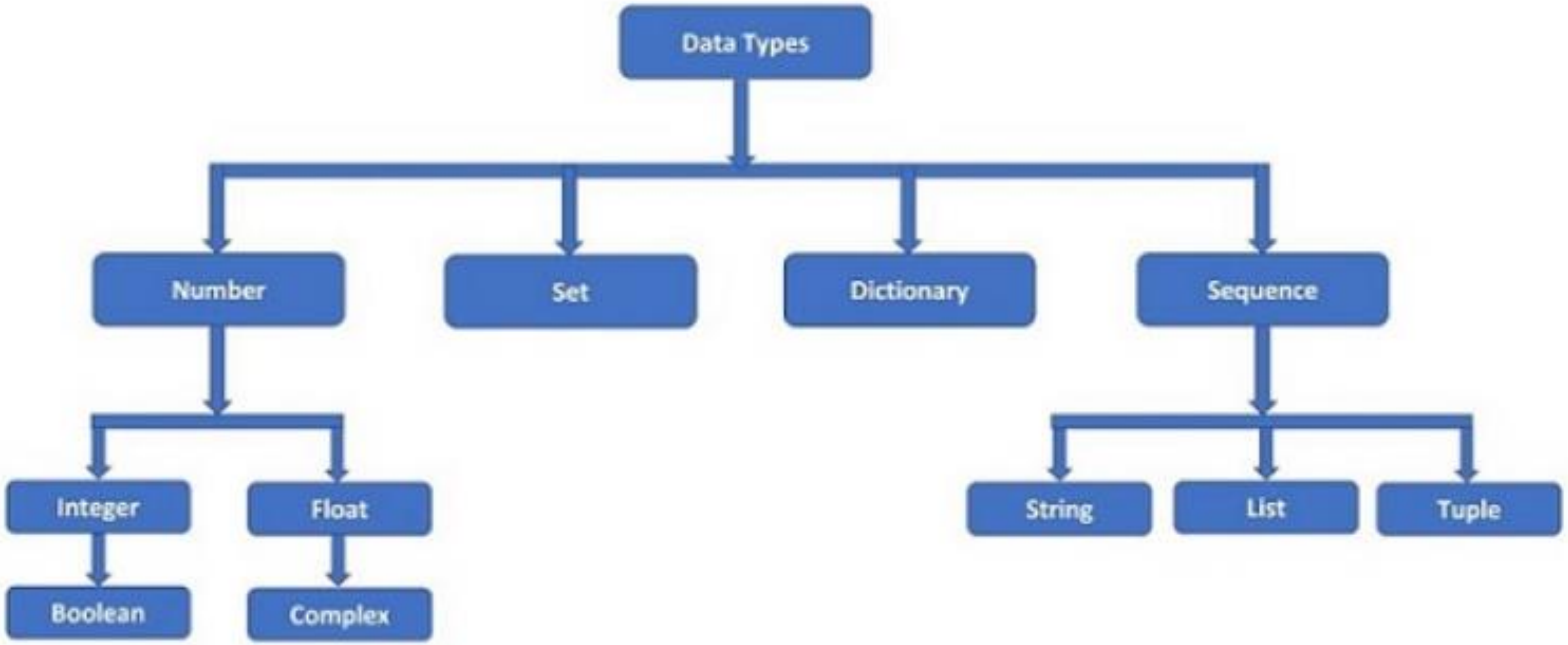
Python Data Types are used to define the type of a variable. It defines what type of data we are going to store in a variable. The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Types of Python Data Types

Python has the following built-in data types which we will discuss in this tutorial:

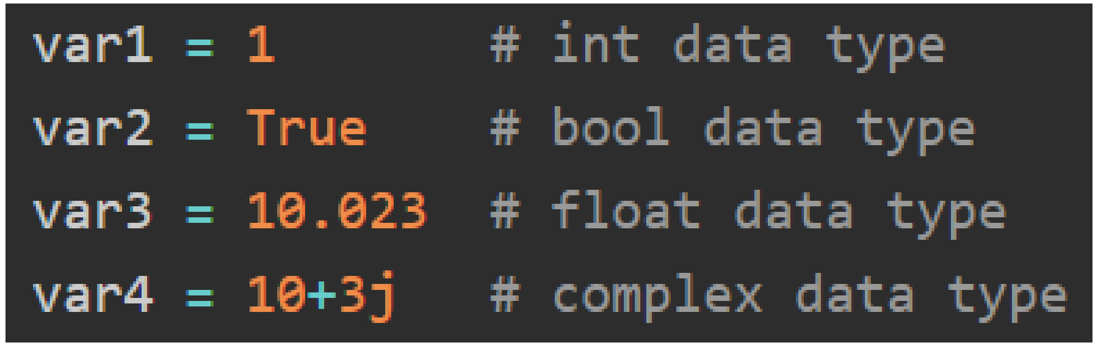


Data Type	Examples
Numeric	int, float, complex
String	str (text sequence type)
Sequence	list, tuple, range
Binary	bytes, bytearray, memoryview
Mapping	dict
Boolean	bool
Set	set, frozenset
None	NoneType


Data Types
Number
Set
Dictionary
Sequence
Integer
Float
String
List
Tuple
Boolean
Complex

Python Numeric Data Type

Python numeric data types store numeric values. Number objects are created when you assign a value to them. For example –

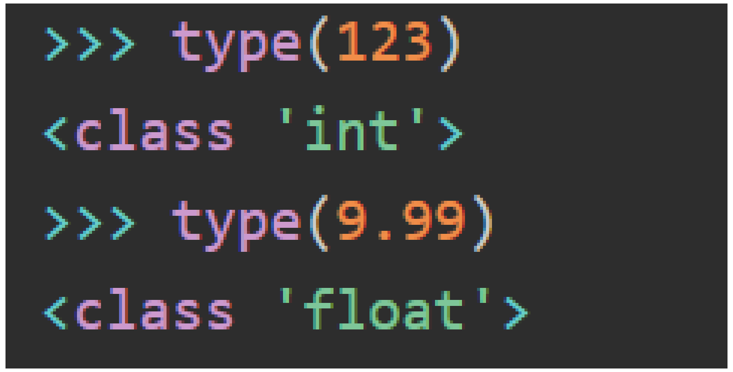


```
var1 = 1        # int data type
var2 = True     # bool data type
var3 = 10.023   # float data type
var4 = 10+3j    # complex data type
```

Python supports four different numerical types and each of them have built-in classes in Python library, called int, bool, float and complex respectively –

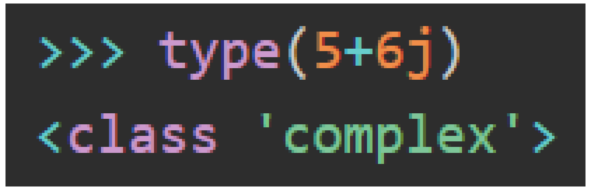
- int (signed integers)
- Bool (boolean)
- float (floating point real values)
- complex (complex numbers)

Python's standard library has a built-in function type(), which returns the class of the given object. Here, it is used to check the type of an integer and floating point number.



```
>>> type(123)
<class 'int'>
>>> type(9.99)
<class 'float'>
```

A complex number is made up of two parts - real and imaginary. They are separated by '+' or '-' signs. The imaginary part is suffixed by 'j' which is the imaginary number. Complex number in Python is represented as x+yj, where x is the real part, and y is the imaginary part. So, 5+6j is a complex number.

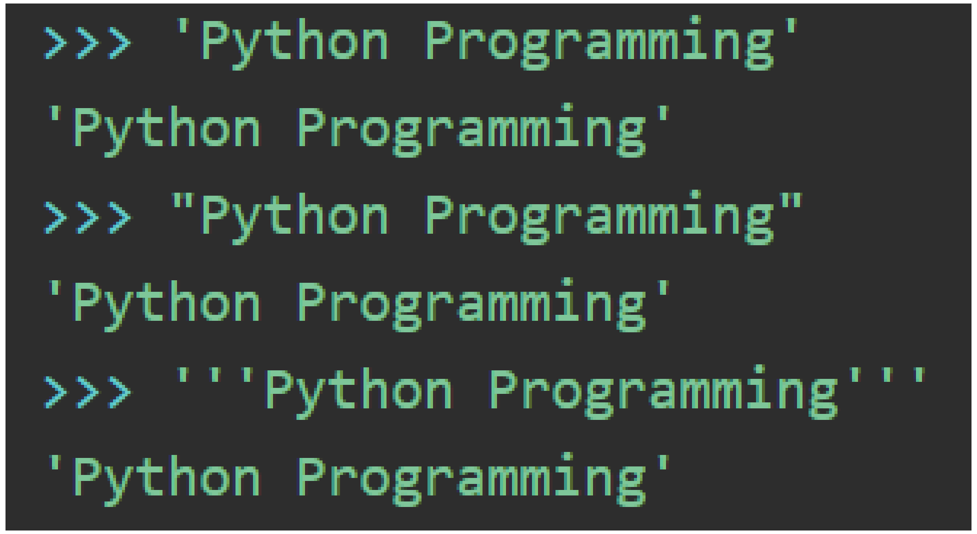


```
>>> type(5+6j)
<class 'complex'>
```

Python String Data Type

Python string is a sequence of one or more Unicode characters, enclosed in single, double or triple quotation marks (also called inverted commas). Python strings are immutable which means when you perform an operation on strings, you always produce a new string object of the same type, rather than mutating an existing string.

As long as the same sequence of characters is enclosed, single or double or triple quotes don't matter. Hence, following string representations are equivalent.



```
>>> 'Python Programming'
'Python Programming'
>>> "Python Programming"
'Python Programming'
>>> '''Python Programming'''
'Python Programming'
```

String

A string is a sequence of characters enclosed by matching single (') or double (") quotes. Ex: "Happy birthday!" and '21' are both strings.

To include a single quote (') in a string, enclose the string with matching double quotes ("). Ex: "Won't this work?" To include a double quote ("), enclose the string with matching single quotes ('). Ex: 'They said "Try it!", so I did'.

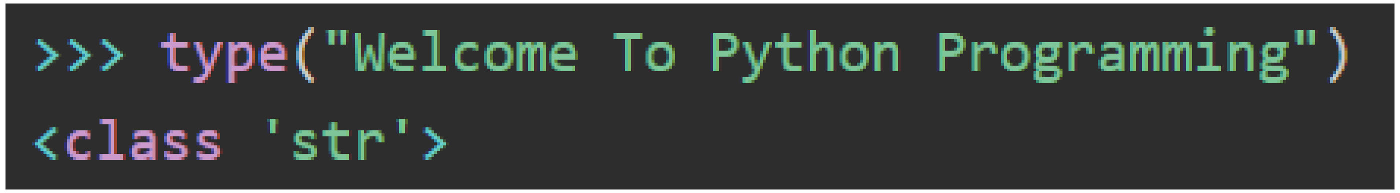
len() function

A common operation on a string object is to get the string length, or the number of characters in the string. The len() function, when called on a string value, returns the string length.

Concatenation

Concatenation is an operation that combines two or more strings sequentially with the concatenation operator (+). Ex: "A" + "part" produces the string "Apart".

A string in Python is an object of str class. It can be verified with type() function.



```
>>> type("Welcome To Python Programming")
<class 'str'>
```

A string is a non-numeric data type. Obviously, we cannot perform arithmetic operations on it. However, operations such as slicing and concatenation can be done. Python's str class defines a number of useful methods for string processing. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator in Python.

Example of String Data Type

```
str = 'Hello World!'

print (str)          # Prints complete string
print (str[0])       # Prints first character of the string
print (str[2:5])     # Prints characters starting from 3rd to 5th
print (str[2:])      # Prints string starting from 3rd character
print (str * 2)      # Prints string two times
print (str + "TEST") # Prints concatenated string
```

Result:

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Valid string	Invalid string
"17" or '17'	17
"seventeen" or 'seventeen'	seventeen
"Where?" or 'Where?'	"Where?'
"I hope you aren't sad."	'I hope you aren't sad.'
'The teacher said "Correct!" '	"The teacher said "Correct!" "

Python Sequence Data Types

Sequence is a collection data type. It is an **ordered** collection of items. Items in the sequence have a positional index starting with 0. It is conceptually similar to an array in C or C++. There are following three sequence data types defined in Python.

- **List Data Type**
- **Tuple Data Type**
- **Range Data Type**

Python sequences are bounded and iterable - Whenever we say an iterable in Python, it means a sequence data type (for example, a list).

Python List Data Type

Python Lists are the most versatile compound data types. A Python list contains items separated by commas and enclosed within square brackets ([]). To some extent, Python lists are similar to arrays in C. One difference between them is that all the items belonging to a Python list can be of **different data type** where as C array can store elements related to a particular data type.

```
>>> [2023, "Python", 3.11, 5+6j, 1.23E-4]
```

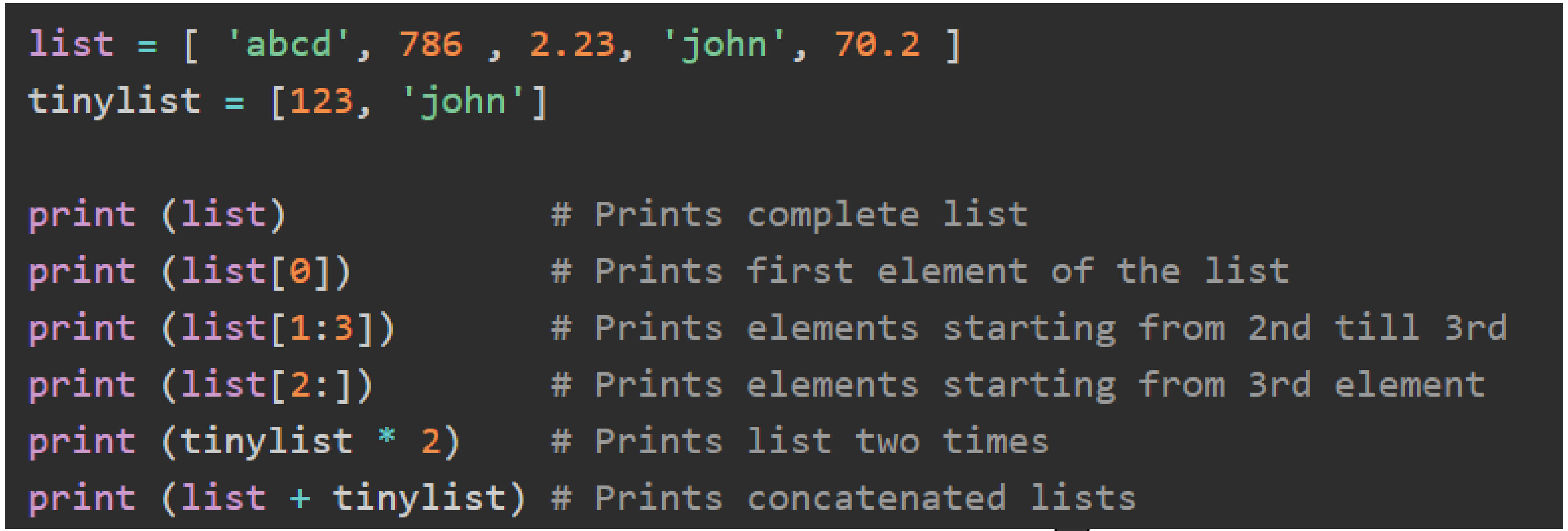
A list in Python is an object of list class. We can check it with type() function.

```
>>> type([2023, "Python", 3.11, 5+6j, 1.23E-4])
<class 'list'>
```

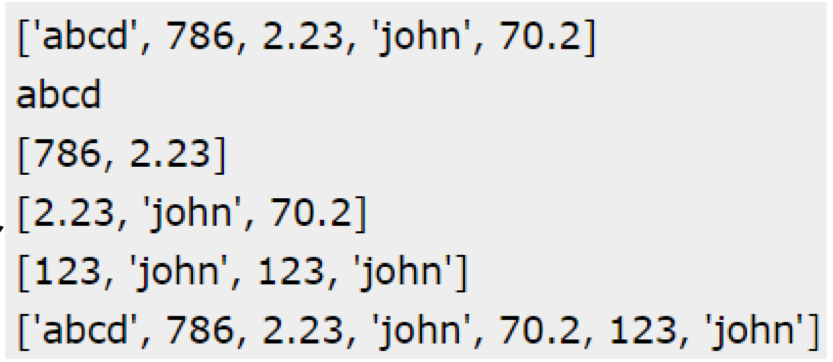
As mentioned, an item in the list may be of any data type. It means that a list object can also be an item in another list. In that case, it becomes a nested list.

```
>>> [['One', 'Two', 'Three'], [1,2,3], [1.0, 2.0, 3.0]]
```

The values stored in a Python list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.



```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print (list)            # Prints complete list
print (list[0])         # Prints first element of the list
print (list[1:3])       # Prints elements starting from 2nd till 3rd
print (list[2:])        # Prints elements starting from 3rd element
print (tinylist * 2)    # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

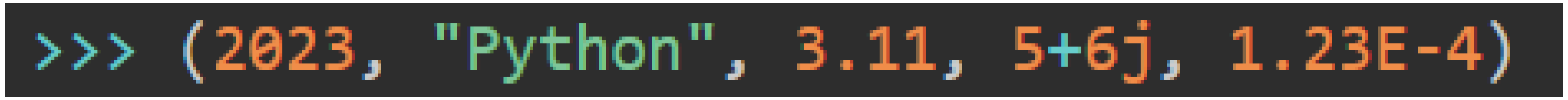


```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuple Data Type

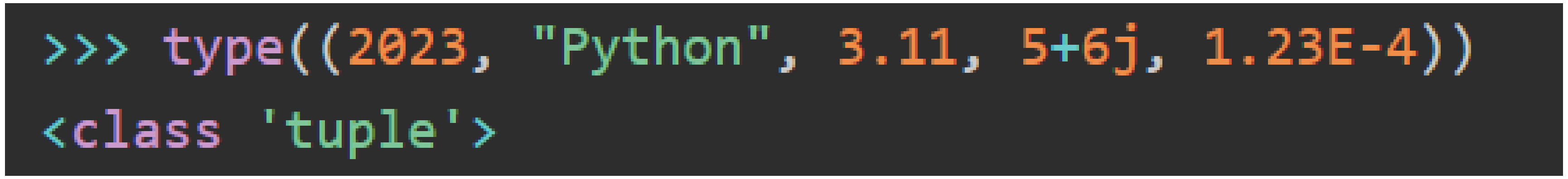
Python tuple is another sequence data type that is similar to a list. A Python tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses (...).

A tuple is also a sequence, hence each item in the tuple has an index referring to its position in the collection. The index starts from 0.



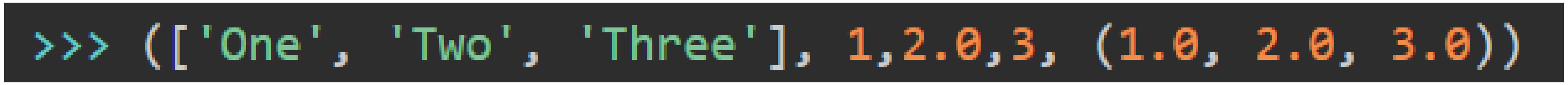
```
>>> (2023, "Python", 3.11, 5+6j, 1.23E-4)
```

In Python, a tuple is an object of tuple class. We can check it with the type() function.



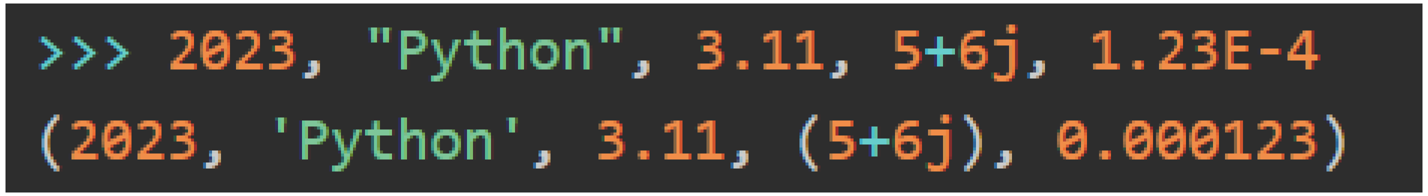
```
>>> type((2023, "Python", 3.11, 5+6j, 1.23E-4))
<class 'tuple'>
```

As in case of a list, an item in the tuple may also be a list, a tuple itself or an object of any other Python class.



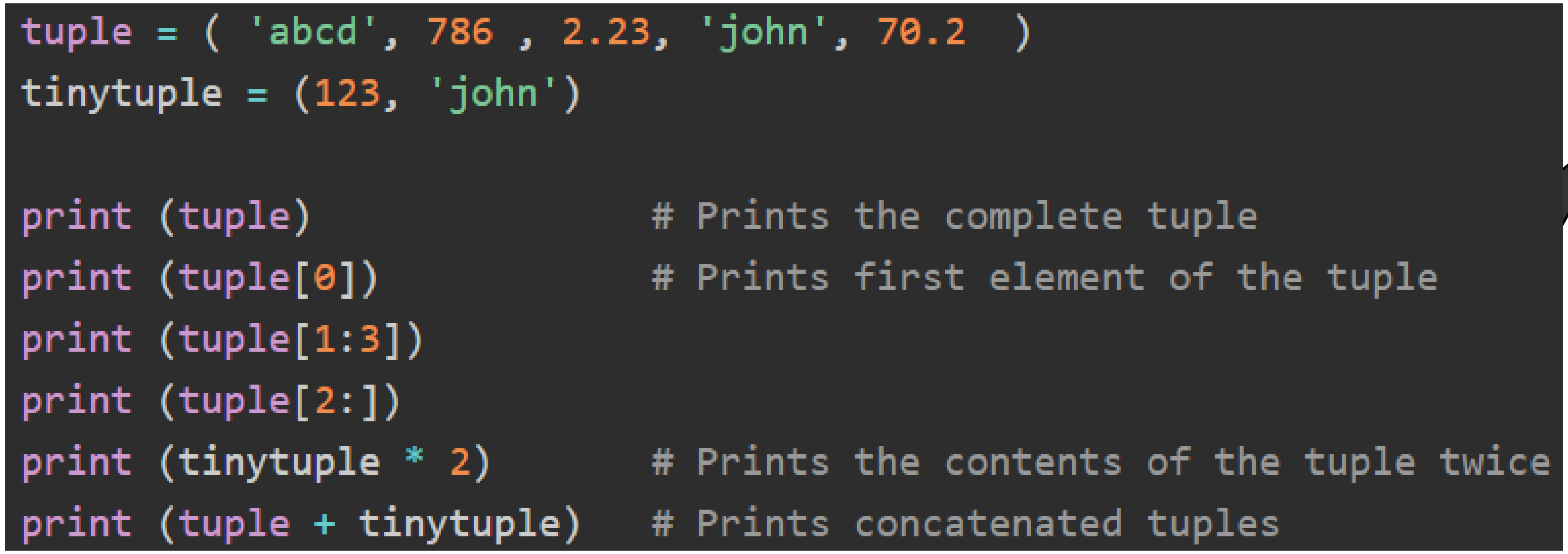
```
>>> (['One', 'Two', 'Three'], 1,2.0,3, (1.0, 2.0, 3.0))
```

To form a tuple, use of parentheses is optional. Data items separated by comma without any enclosing symbols are treated as a tuple by default.

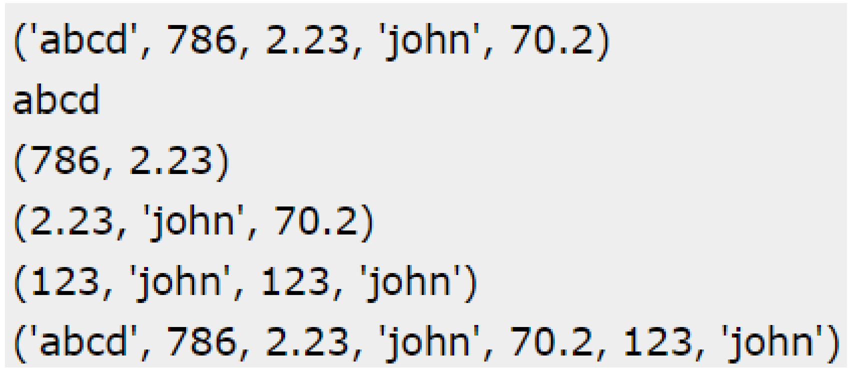


```
>>> 2023, "Python", 3.11, 5+6j, 1.23E-4
(2023, 'Python', 3.11, (5+6j), 0.000123)
```

Example of Tuple data Type



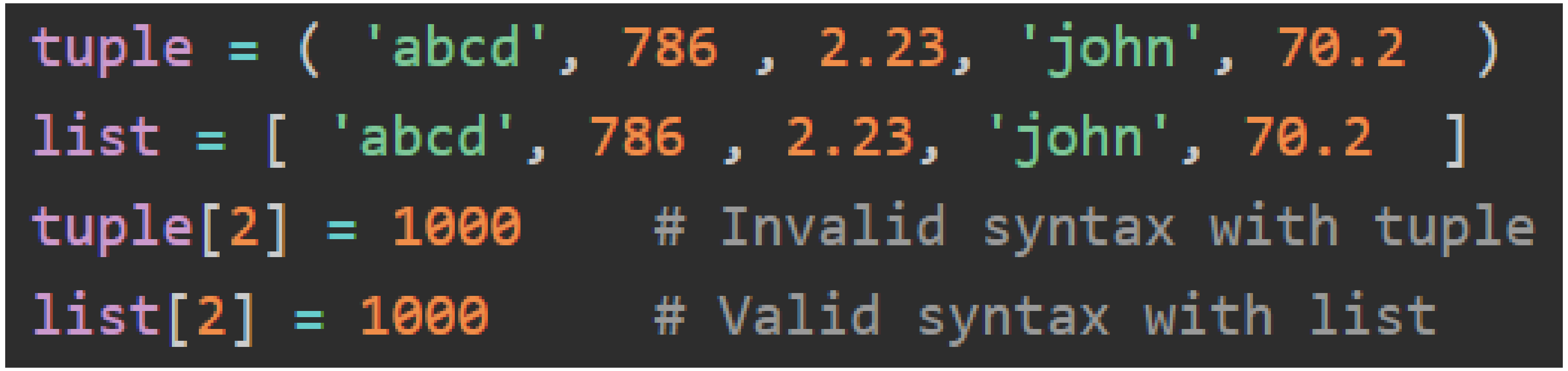
```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print (tuple)               # Prints the complete tuple
print (tuple[0])            # Prints first element of the tuple
print (tuple[1:3])
print (tuple[2:])
print (tinytuple * 2)       # Prints the contents of the tuple twice
print (tuple + tinytuple)   # Prints concatenated tuples
```



```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The main differences between lists and tuples are: **Lists are enclosed in brackets ([ ])** and their elements and size can be changed i.e. lists are mutable, while **tuples are enclosed in parentheses (( ))** and cannot be updated (immutable). **Tuples can be thought of as read-only lists.**

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

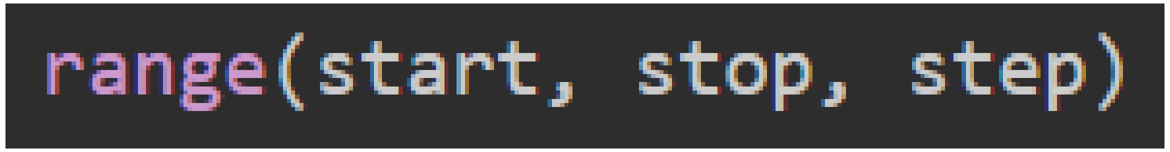


```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Range Data Type

A Python range is an immutable sequence of numbers which is typically used to iterate through a specific number of items.

It is represented by the **Range class**. The constructor of this class accepts a sequence of numbers starting from 0 and increments to 1 until it reaches a specified number. Following is the syntax of the function −



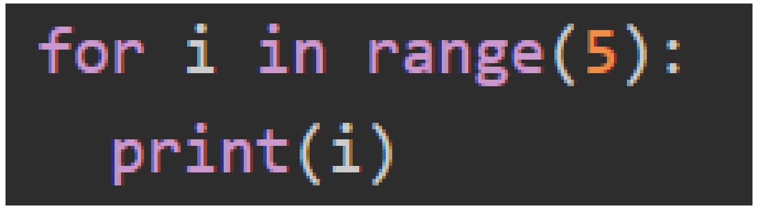
```
range(start, stop, step)
```

Here is the description of the parameters used −

- **start:** Integer number to specify starting position, (Its optional, Default: 0)
- **stop:** Integer number to specify ending position (It's mandatory)
- **step:** Integer number to specify increment, (Its optional, Default: 1)

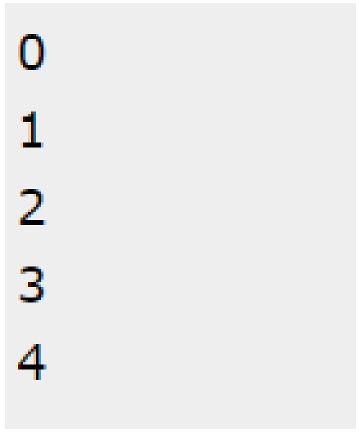
Example of Range Data Type

Following is a program which uses for loop to print number from 0 to 4 –



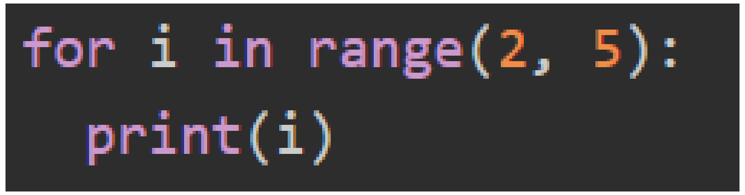
```python
for i in range(5):
  print(i)
```

Result:



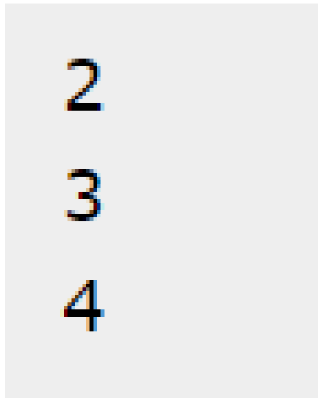
```
0
1
2
3
4
```

Now let's modify above program to print the number starting from 2 instead of 0 –



```python
for i in range(2, 5):
  print(i)
```

Result:



```
2
3
4
```

Again, let's modify the program to print the number starting from 1 but with an increment of 2 instead of 1:

```
for i in range(1, 5, 2):
  print(i)
```

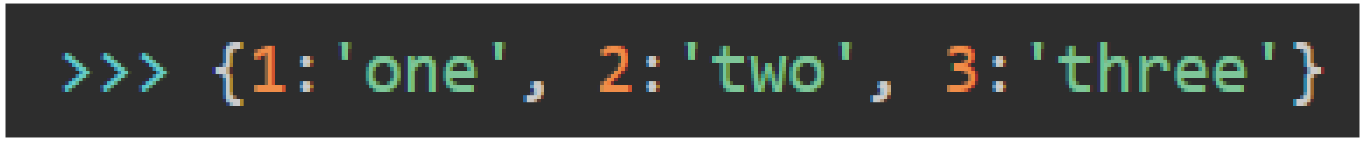
This produce the following result −

```
1
3
```

Python Dictionary Data Type

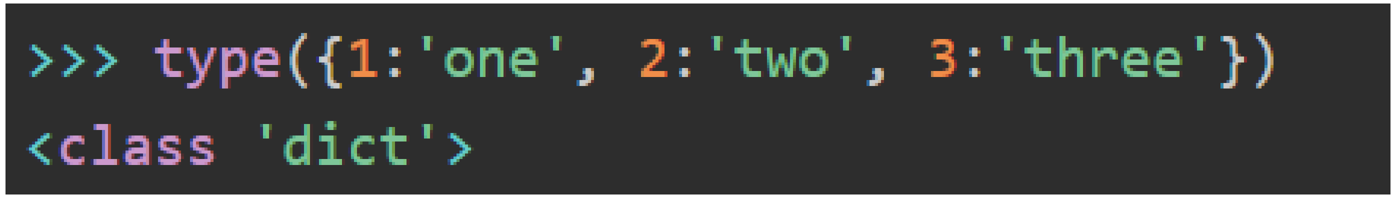
Python dictionaries are kind of hash table type. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Python dictionary is like associative arrays or hashes found in Perl and consist of **key:value pairs**. The pairs are separated by comma and put inside curly brackets {}. To establish mapping between key and value, the semicolon':' symbol is put between the two.



```
>>> {1:'one', 2:'two', 3:'three'}
```

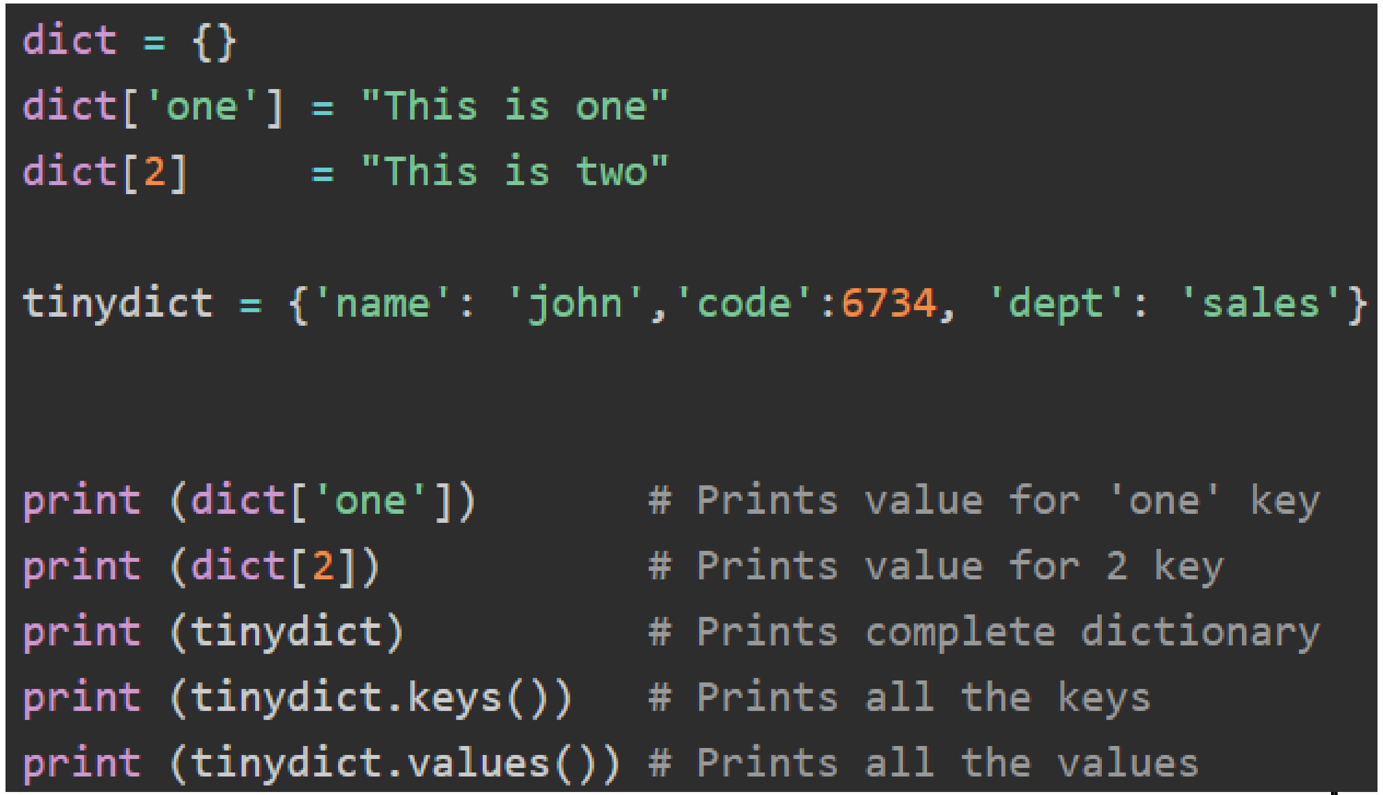
In Python, dictionary is an object of the built-in **dict class**. We can check it with the type() function.



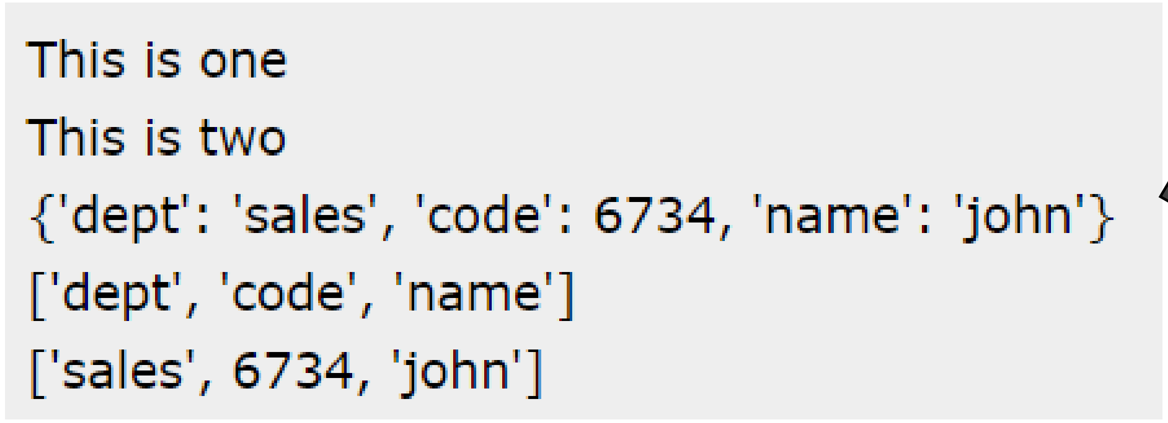
```
>>> type({1:'one', 2:'two', 3:'three'})
<class 'dict'>
```

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

Example of Dictionary Data Type



```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print (dict['one'])       # Prints value for 'one' key
print (dict[2])           # Prints value for 2 key
print (tinydict)          # Prints complete dictionary
print (tinydict.keys())   # Prints all the keys
print (tinydict.values()) # Prints all the values
```



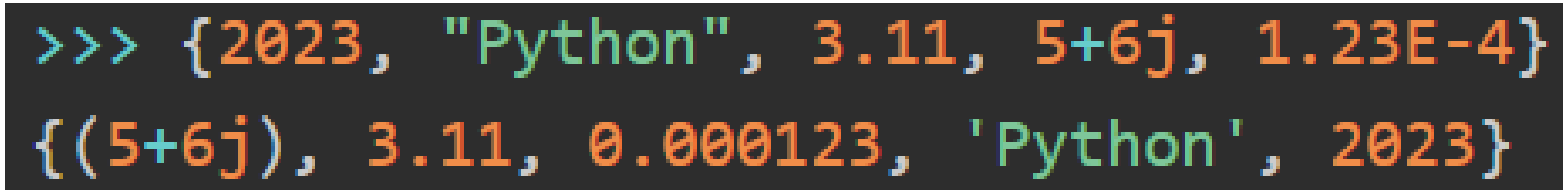
```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Python's dictionary is not a sequence. It is a collection of items but each item (key:value pair) is not identified by positional index as in string, list or tuple. Hence, slicing operation cannot be done on a dictionary. Dictionary is a mutable object, so it is possible to perform add, modify or delete actions with corresponding functionality defined in dict class. These operations will be explained in a subsequent chapter.

Python Set Data Type

Set is a Python implementation of set as defined in Mathematics. A set in Python is a collection, but is not an indexed or ordered collection as string, list or tuple. **An object cannot appear more than once in a set**, whereas in List and Tuple, same object can appear more than once.

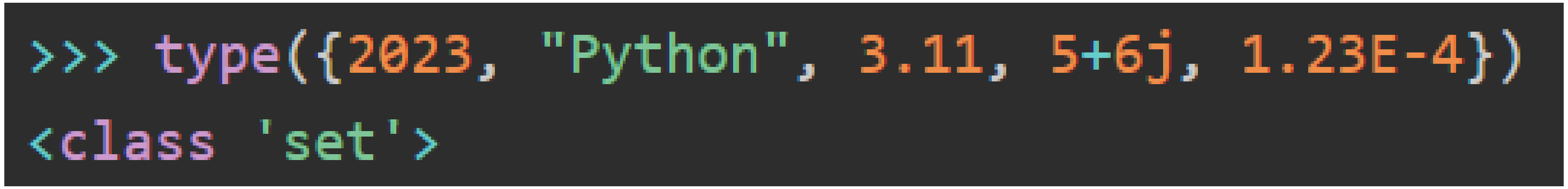
Comma separated items in a set are put inside curly brackets or braces {}. Items in the set collection can be of different data types.



```
>>> {2023, "Python", 3.11, 5+6j, 1.23E-4}
{(5+6j), 3.11, 0.000123, 'Python', 2023}
```

Note that items in the set collection may not follow the same order in which they are entered. The position of items is optimized by Python to perform operations over set as defined in mathematics.

Python's Set is an object of built-in set class, as can be checked with the type() function.

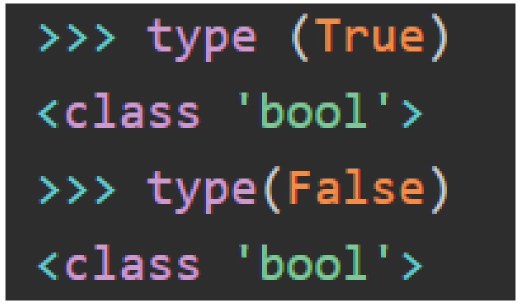


```
>>> type({2023, "Python", 3.11, 5+6j, 1.23E-4})
<class 'set'>
```

Python Boolean Data Type

Python boolean type is one of built-in data types which represents <u>one of the two values either **True or False**</u>. Python bool() function allows you to evaluate the value of any expression and returns either True or False based on the expression.

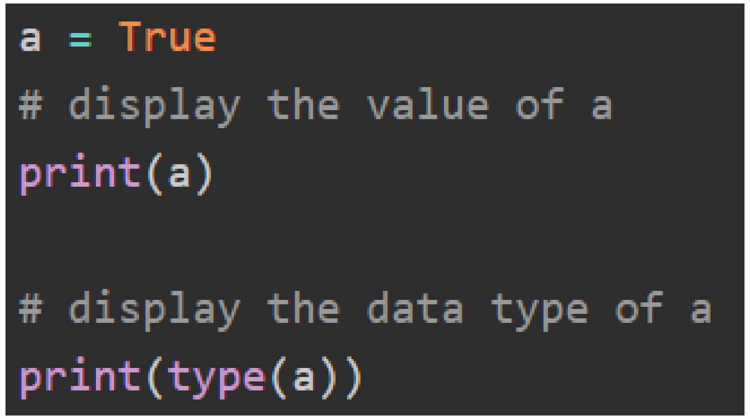
A Boolean number has only two possible values, as represented by the keywords, <u>True and False</u>. They correspond to integer 1 and 0 respectively.



```
>>> type (True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

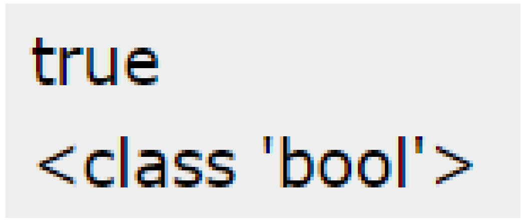
Example of Boolean Data Type

Following is a program which prints the value of boolean variables a and b –



```
a = True
# display the value of a
print(a)

# display the data type of a
print(type(a))
```

Result:



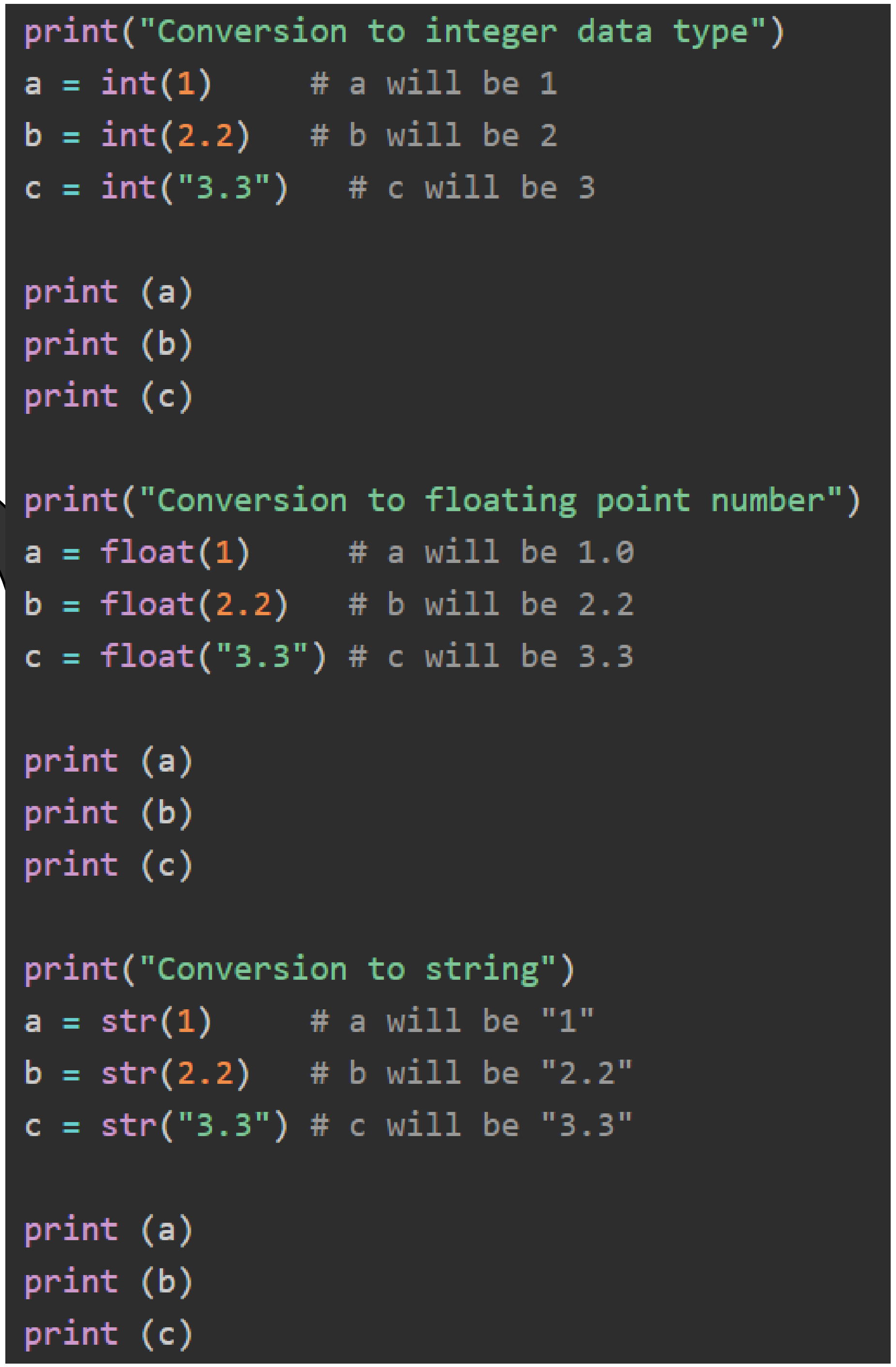
```
true
<class 'bool'>
```

Python Data Type Conversion

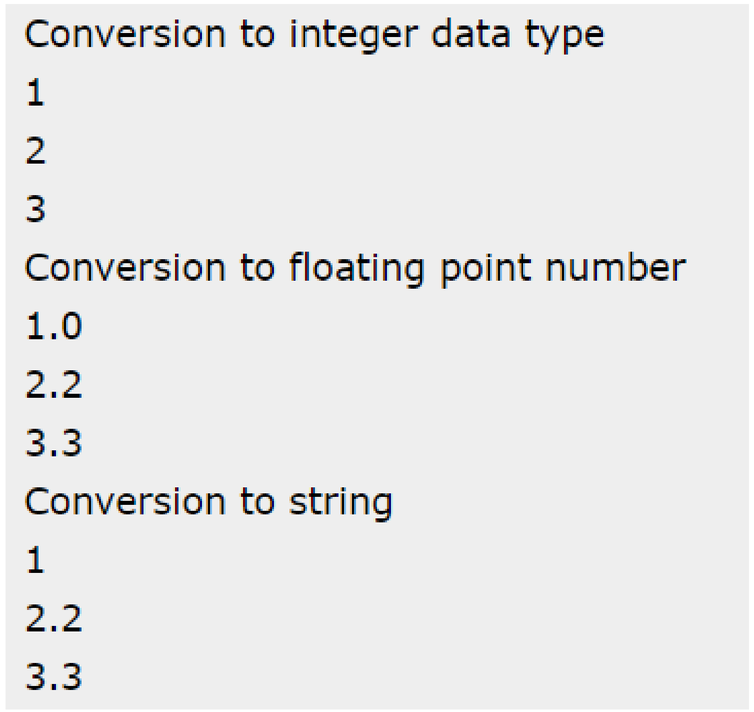
Sometimes, you may need to perform conversions between the built-in data types. To convert data between different Python data types, you simply use the type name as a function.

Example

Following is an example which converts different values to integer, floating point and string values respectively –



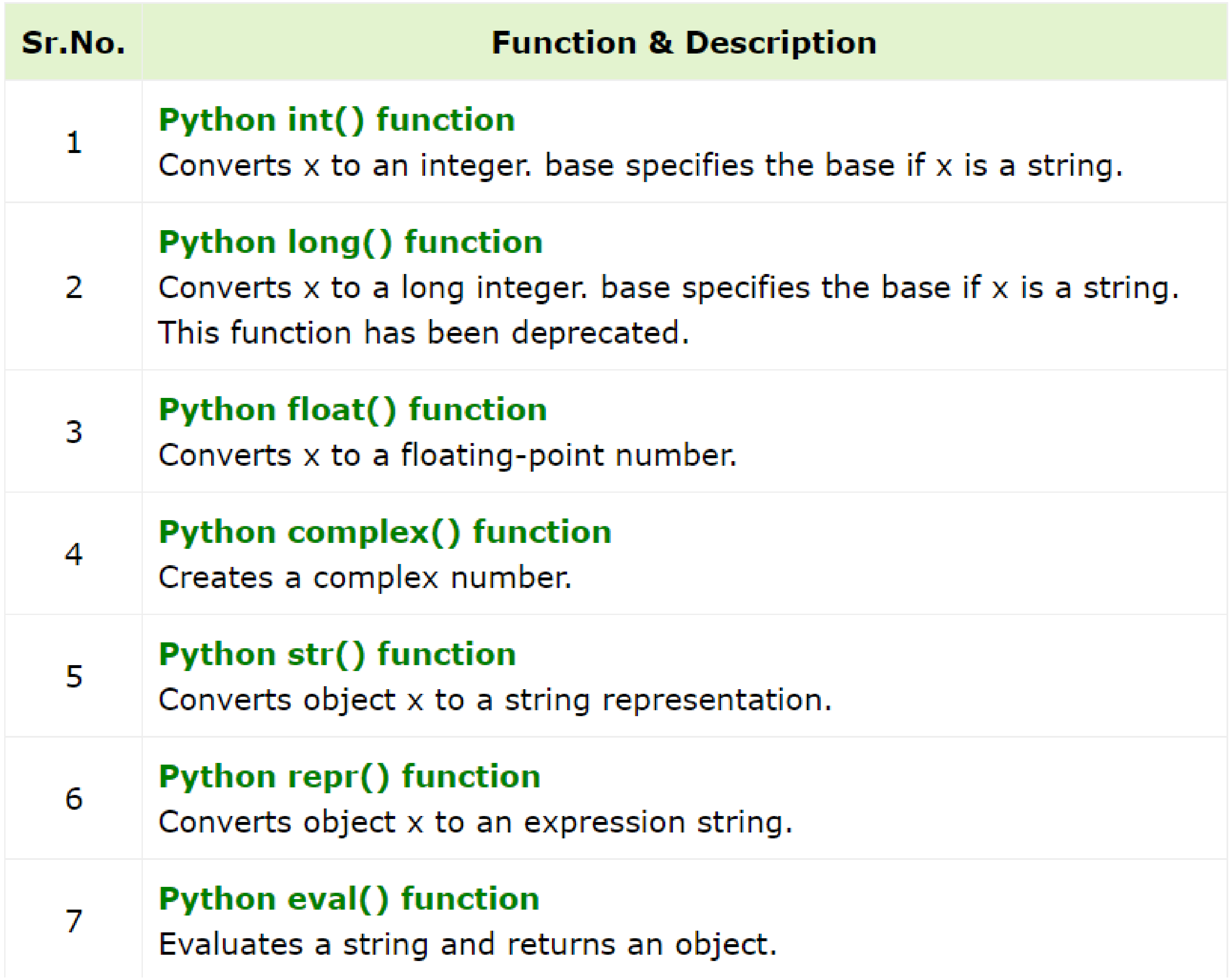
```python
print("Conversion to integer data type")
a = int(1)     # a will be 1
b = int(2.2)   # b will be 2
c = int("3.3")   # c will be 3

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

print("Conversion to floating point number")
a = float(1)     # a will be 1.0
b = float(2.2)   # b will be 2.2
c = float("3.3") # c will be 3.3

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

print("Conversion to string")
a = str(1)     # a will be "1"
b = str(2.2)   # b will be "2.2"
c = str("3.3") # c will be "3.3"

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



```
Conversion to integer data type
1
2
3
Conversion to floating point number
1.0
2.2
3.3
Conversion to string
1
2.2
3.3
```

Data Type Conversion Functions

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.



Sr.No.	Function & Description
1	**Python int() function** Converts x to an integer. base specifies the base if x is a string.
2	**Python long() function** Converts x to a long integer. base specifies the base if x is a string. This function has been deprecated.
3	**Python float() function** Converts x to a floating-point number.
4	**Python complex() function** Creates a complex number.
5	**Python str() function** Converts object x to a string representation.
6	**Python repr() function** Converts object x to an expression string.
7	**Python eval() function** Evaluates a string and returns an object.

8	**Python tuple() function** Converts s to a tuple.
9	**Python list() function** Converts s to a list.
10	**Python set() function** Converts s to a set.
11	**Python dict() function** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**Python frozenset() function** Converts s to a frozen set.
13	**Python chr() function** Converts an integer to a character.
14	**Python unichr() function** Converts an integer to a Unicode character.
15	**Python ord() function** Converts a single character to its integer value.
16	**Python hex() function** Converts an integer to a hexadecimal string.
17	**Python oct() function** Converts an integer to an octal string.

Basic output

The print() function displays output to the user. Output is the information or result produced by a program. The **sep** and **end** options can be used to customize the output.

- Multiple values, separated by commas, can be printed in the same statement. By default, each value is separated by a space character in the output. The sep option can be used to change this behavior.
- By default, the print() function adds a newline character at the end of the output. A newline character tells the display to move to the next line. The end option can be used to continue printing on the same line.

Basic input

Computer programs often receive input from the user. Input is what a user enters into a program. An input statement, variable = input("prompt"), has three parts:

- A variable refers to a value stored in memory. In the statement above, variable can be replaced with any name the programmer chooses.
- The input() function reads one line of input from the user. A function is a named, reusable block of code that performs a task when called. The input is stored in the computer's memory and can be accessed later using the variable.
- A prompt is a short message that indicates the program is waiting for input. In the statement above, "prompt" can be omitted or replaced with any message.

Code	Output

`print("Today is Monday.")`
`print("I like string beans.")`
	`Today is Monday.`
	`I like string beans.`

`print("Today", "is", "Monday")`
`print("Today", "is", "Monday", sep="...")`
	`Today is Monday`
	`Today...is...Monday`

`print("Today is Monday, ", end="")`
`print("I like string beans.")`
	`Today is Monday, I like string`
	`beans.`

`print("Today", "is", "Monday", sep="? ",`
`end="!!")`
`print("I like string beans.")`
	`Today? is? Monday!!I like string`
	`beans.`

Function	Description
`print(values)`	Outputs one or more values, each separated by a space, to the user.
`input(prompt)`	If present, `prompt` is output to the user. The function then reads a line of input from the user.
`len(string)`	Returns the length (the number of characters) of a string.
`type(value)`	Returns the type (or class) of a value. Ex: `type(123)` is `<class 'int'>`.

Syntax	Description
# (Comment)	All text is ignored from the # symbol to the end of the line.
' or " (String)	Strings may be written using either kind of quote. Ex: `'A'` and `"A"` represent the same string. By convention, this book uses double quotes (") for most strings.
""" (Docstring)	Used for documentation, often in multi-line strings, to summarize a program's purpose or usage.

Python Operators

Python operators are special symbols (sometimes called keywords) that are used to perform certain most commonly required operations on one or more operands (value, variables, or expressions).

Types of Operators in Python

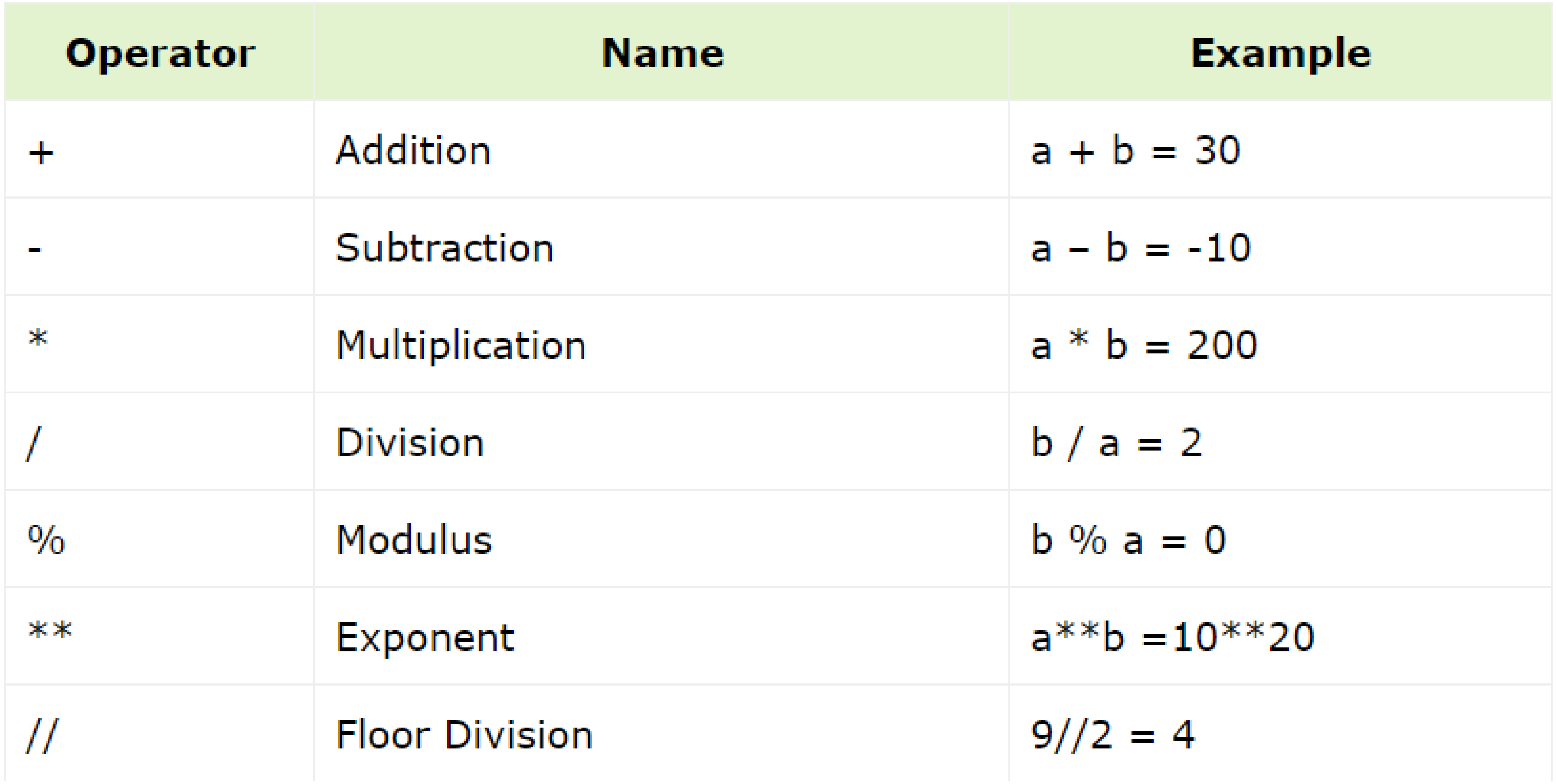
Python language supports the following types of operators −

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Python Arithmetic Operators

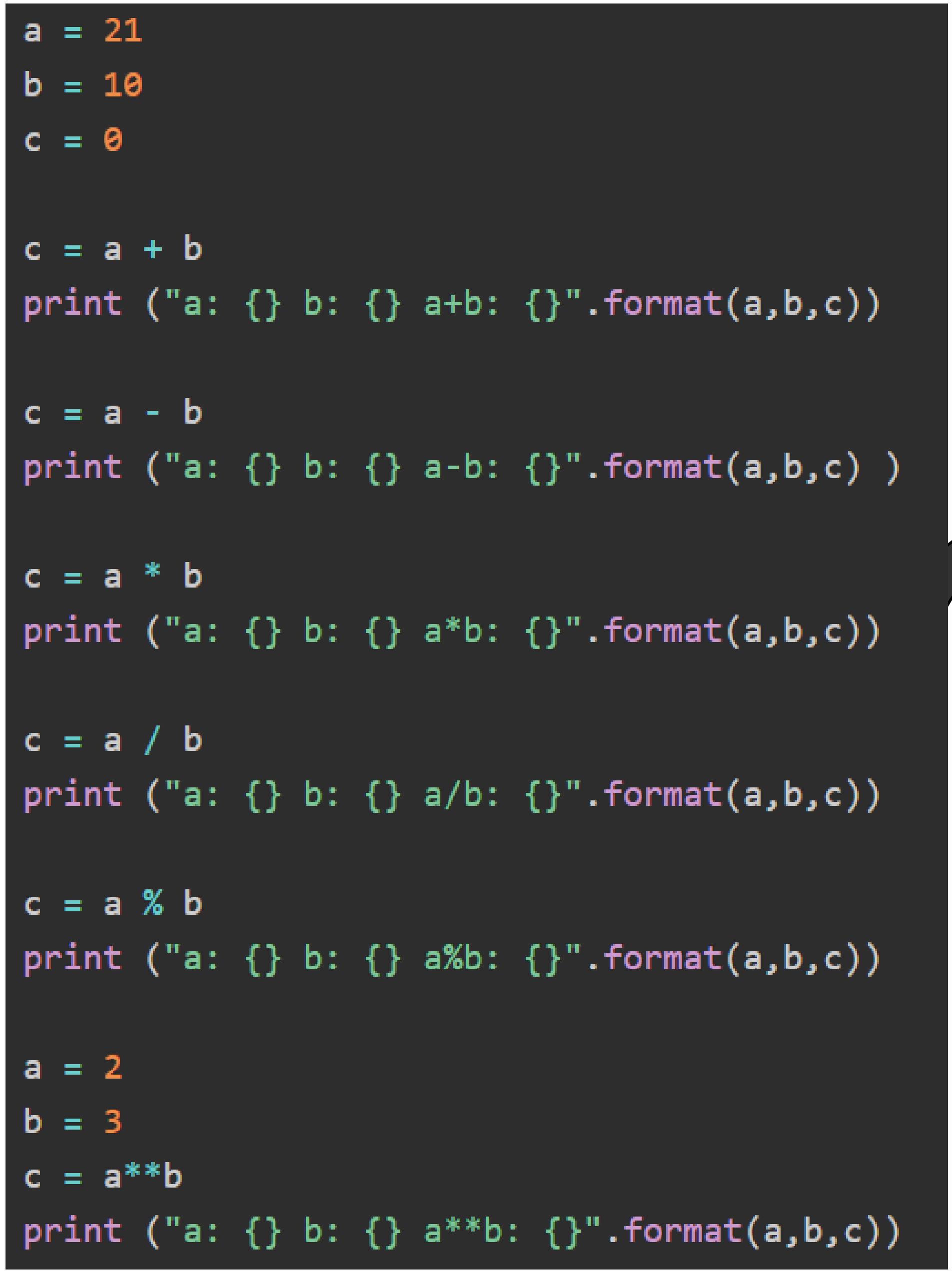
Arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, etc.

Assume variable a holds 10 and variable b holds 20, then

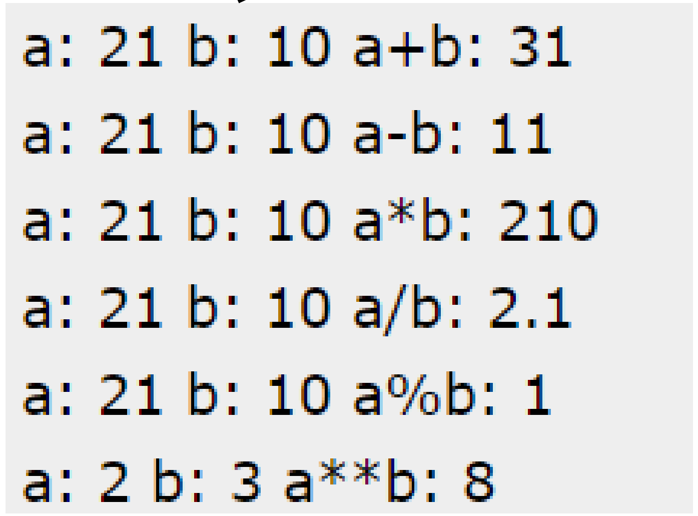


Operator	Name	Example
+	Addition	a + b = 30
-	Subtraction	a – b = -10
*	Multiplication	a * b = 200
/	Division	b / a = 2
%	Modulus	b % a = 0
**	Exponent	a**b =10**20
//	Floor Division	9//2 = 4

Example of Python Arithmetic Operators



```python
a = 21
b = 10
c = 0

c = a + b
print ("a: {} b: {} a+b: {}".format(a,b,c))

c = a - b
print ("a: {} b: {} a-b: {}".format(a,b,c) )

c = a * b
print ("a: {} b: {} a*b: {}".format(a,b,c))

c = a / b
print ("a: {} b: {} a/b: {}".format(a,b,c))

c = a % b
print ("a: {} b: {} a%b: {}".format(a,b,c))

a = 2
b = 3
c = a**b
print ("a: {} b: {} a**b: {}".format(a,b,c))
```



```
a: 21 b: 10 a+b: 31
a: 21 b: 10 a-b: 11
a: 21 b: 10 a*b: 210
a: 21 b: 10 a/b: 2.1
a: 21 b: 10 a%b: 1
a: 2 b: 3 a**b: 8
```

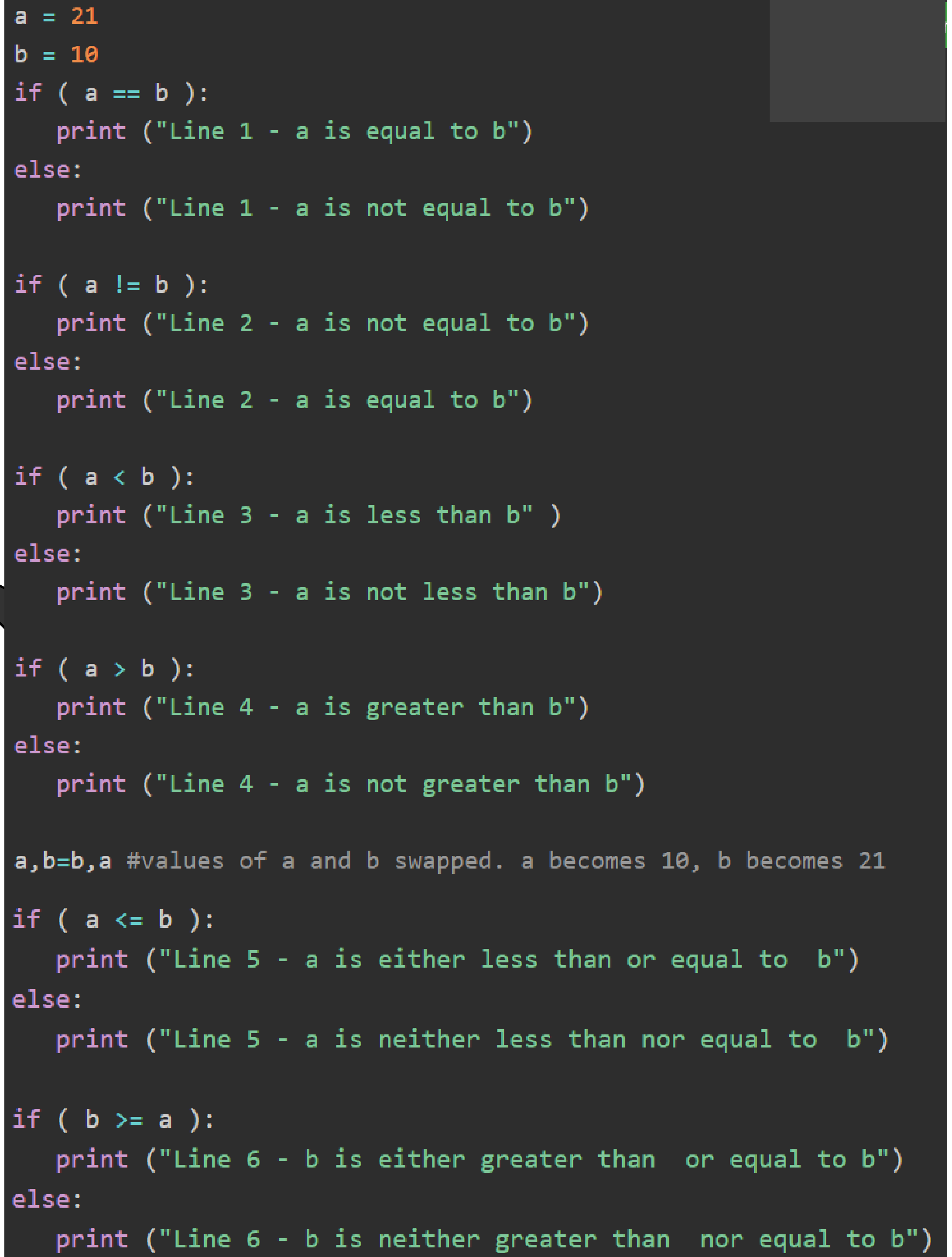
Python Comparison / Rational Operators

Comparison operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

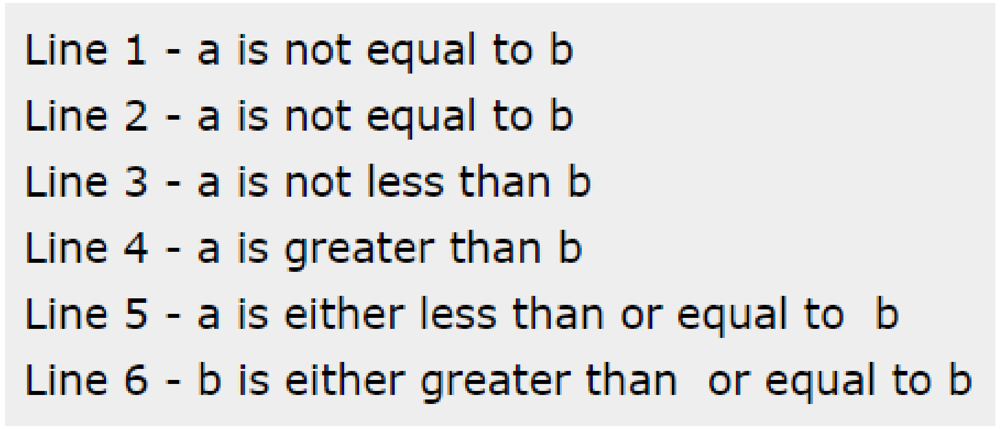
Assume variable a holds 10 and variable b holds 20, then

Operator	Name	Example
==	Equal	(a == b) is not true.
!=	Not equal	(a != b) is true.
>	Greater than	(a > b) is not true.
<	Less than	(a < b) is true.
>=	Greater than or equal to	(a >= b) is not true.
<=	Less than or equal to	(a <= b) is true.

Example of Python Comparison Operators



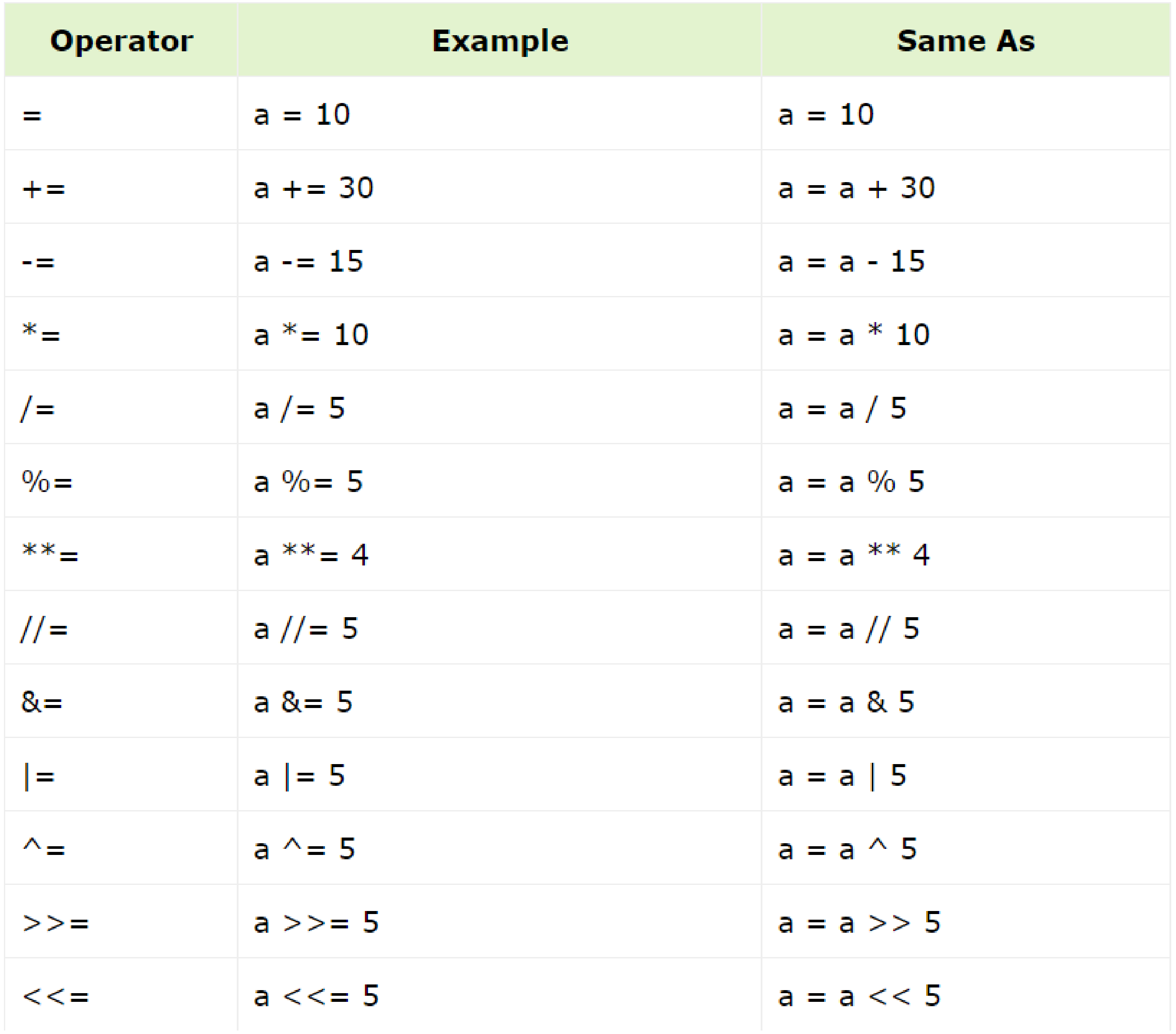
```python
a = 21
b = 10
if ( a == b ):
   print ("Line 1 - a is equal to b")
else:
   print ("Line 1 - a is not equal to b")

if ( a != b ):
   print ("Line 2 - a is not equal to b")
else:
   print ("Line 2 - a is equal to b")

if ( a < b ):
   print ("Line 3 - a is less than b" )
else:
   print ("Line 3 - a is not less than b")

if ( a > b ):
   print ("Line 4 - a is greater than b")
else:
   print ("Line 4 - a is not greater than b")

a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21

if ( a <= b ):
   print ("Line 5 - a is either less than or equal to  b")
else:
   print ("Line 5 - a is neither less than nor equal to  b")

if ( b >= a ):
   print ("Line 6 - b is either greater than  or equal to b")
else:
   print ("Line 6 - b is neither greater than  nor equal to b")
```



```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not less than b
Line 4 - a is greater than b
Line 5 - a is either less than or equal to  b
Line 6 - b is either greater than  or equal to b
```

Python Assignment Operators

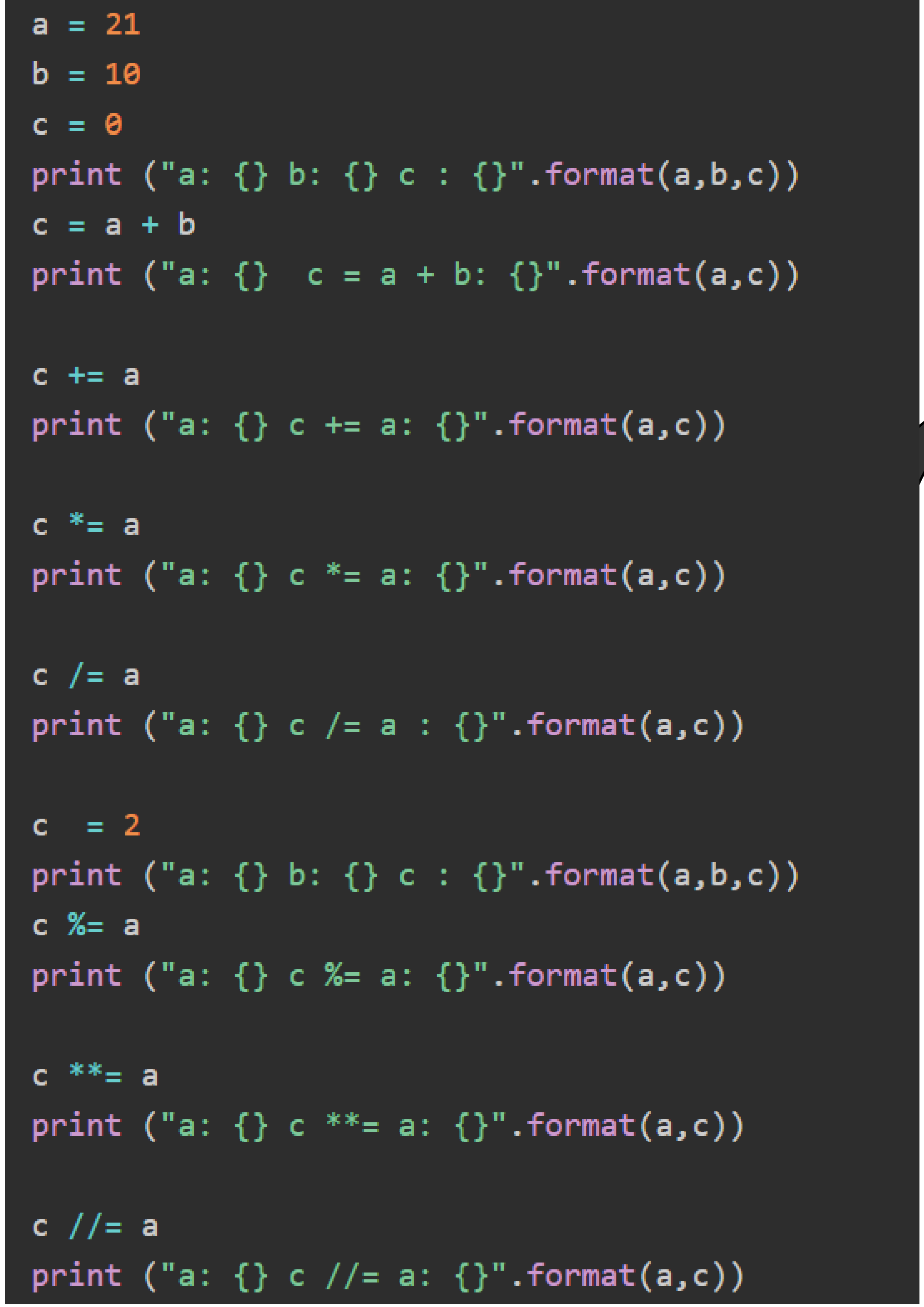
Assignment operators are used to assign values to variables. Following is a table which shows all Python assignment operators.



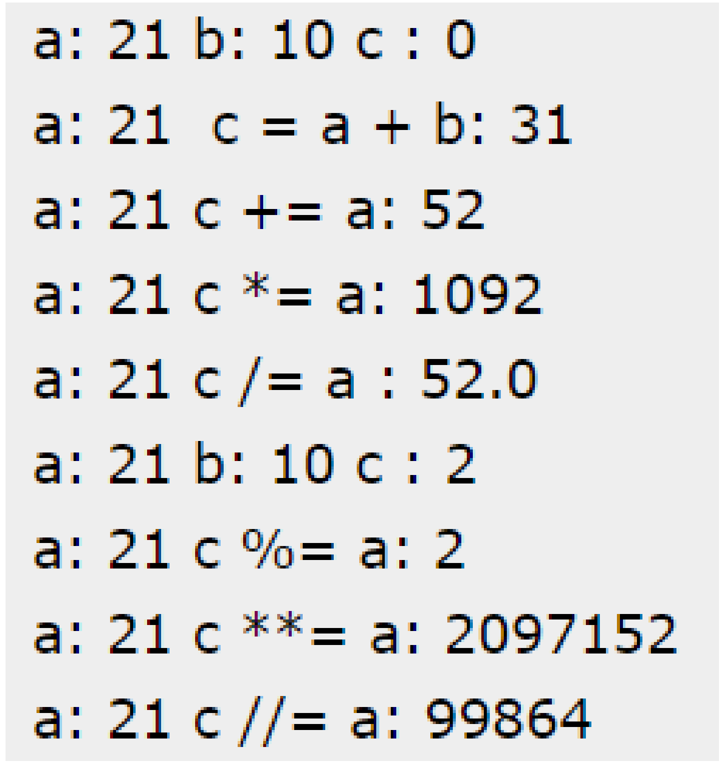
Operator	Example	Same As
=	a = 10	a = 10
+=	a += 30	a = a + 30
-=	a -= 15	a = a - 15
*=	a *= 10	a = a * 10
/=	a /= 5	a = a / 5
%=	a %= 5	a = a % 5
**=	a **= 4	a = a ** 4
//=	a //= 5	a = a // 5
&=	a &= 5	a = a & 5
|=	a |= 5	a = a | 5
^=	a ^= 5	a = a ^ 5
>>=	a >>= 5	a = a >> 5
<<=	a <<= 5	a = a << 5

Operator	Description
= (Assignment)	Assigns (or updates) the value of a variable. In Python, variables begin to exist when assigned for the first time.
+ (Concatenation)	Appends the contents of two strings, resulting in a new string.
+ (Addition)	Adds the values of two numbers.
- (Subtraction)	Subtracts the value of one number from another.
* (Multiplication)	Multiplies the values of two numbers.
/ (Division)	Divides the value of one number by another.
** (Exponentiation)	Raises a number to a power. Ex: `3**2` is three squared.

Example of Python Assignment Operators



```python
a = 21
b = 10
c = 0
print ("a: {} b: {} c : {}".format(a,b,c))
c = a + b
print ("a: {}  c = a + b: {}".format(a,c))

c += a
print ("a: {} c += a: {}".format(a,c))

c *= a
print ("a: {} c *= a: {}".format(a,c))

c /= a
print ("a: {} c /= a : {}".format(a,c))

c  = 2
print ("a: {} b: {} c : {}".format(a,b,c))
c %= a
print ("a: {} c %= a: {}".format(a,c))

c **= a
print ("a: {} c **= a: {}".format(a,c))

c //= a
print ("a: {} c //= a: {}".format(a,c))
```



```
a: 21 b: 10 c : 0
a: 21  c = a + b: 31
a: 21 c += a: 52
a: 21 c *= a: 1092
a: 21 c /= a : 52.0
a: 21 b: 10 c : 2
a: 21 c %= a: 2
a: 21 c **= a: 2097152
a: 21 c //= a: 99864
```

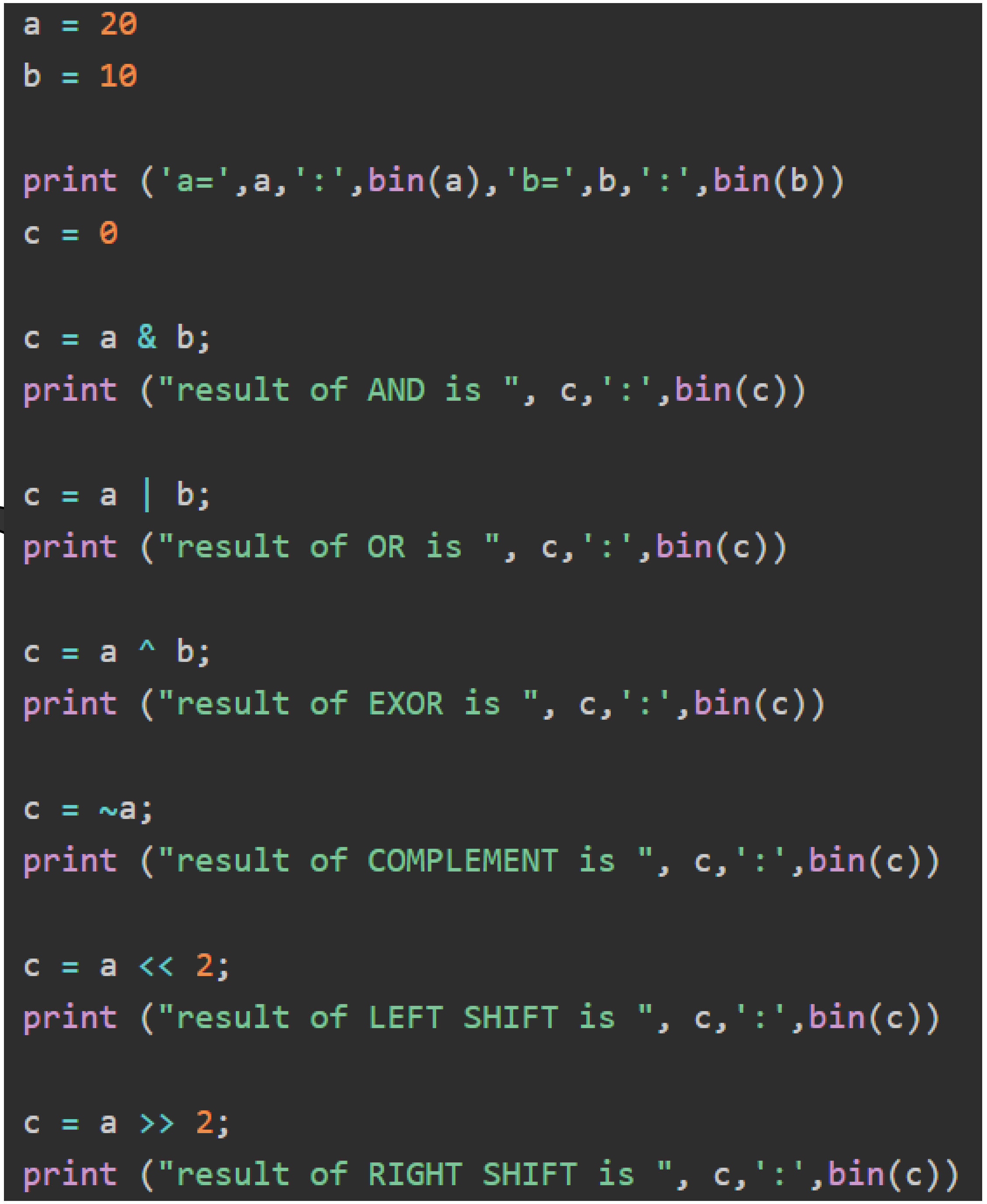
Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. These operators are used to compare binary numbers.

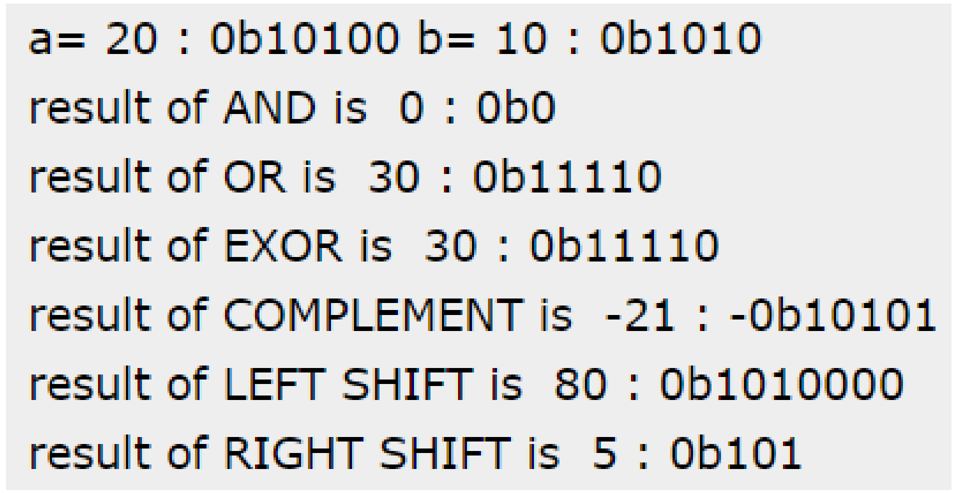
There are following Bitwise operators supported by Python language

Operator	Name	Example
&	AND	a & b
|	OR	a | b
^	XOR	a ^ b
~	NOT	~a
<<	Zero fill left shift	a << 3
>>	Signed right shift	a >> 3

Example of Python Bitwise Operators



```
a = 20
b = 10

print ('a=',a,':',bin(a),'b=',b,':',bin(b))
c = 0

c = a & b;
print ("result of AND is ", c,':',bin(c))

c = a | b;
print ("result of OR is ", c,':',bin(c))

c = a ^ b;
print ("result of EXOR is ", c,':',bin(c))

c = ~a;
print ("result of COMPLEMENT is ", c,':',bin(c))

c = a << 2;
print ("result of LEFT SHIFT is ", c,':',bin(c))

c = a >> 2;
print ("result of RIGHT SHIFT is ", c,':',bin(c))
```



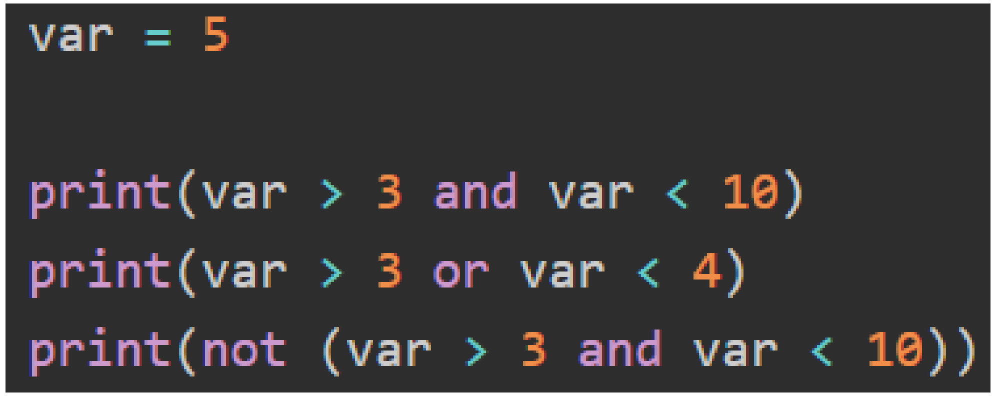
```
a= 20 : 0b10100 b= 10 : 0b1010
result of AND is  0 : 0b0
result of OR is  30 : 0b11110
result of EXOR is  30 : 0b11110
result of COMPLEMENT is  -21 : -0b10101
result of LEFT SHIFT is  80 : 0b1010000
result of RIGHT SHIFT is  5 : 0b101
```

Python Logical Operators

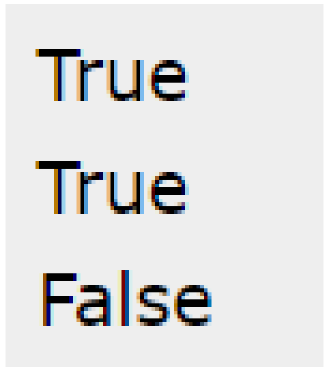
Python logical operators are used to combine two or more conditions and check the final result. There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Name	Example
and	AND	a and b
or	OR	a or b
not	NOT	not(a)

Example of Python Logical Operators



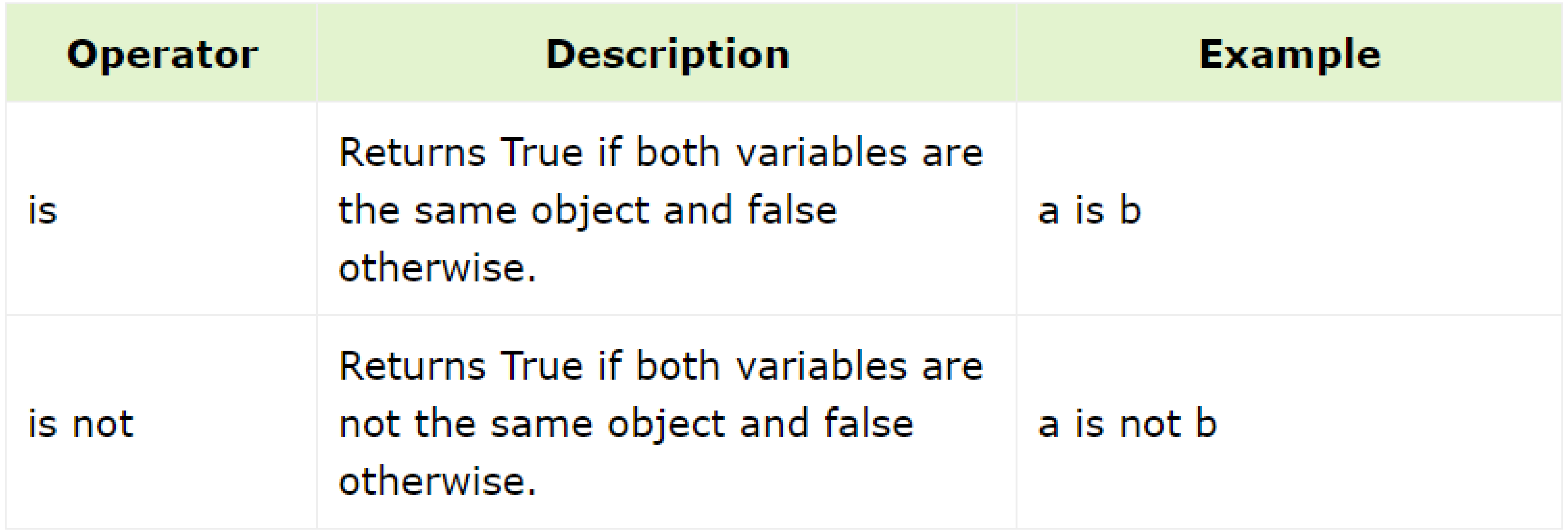
```python
var = 5

print(var > 3 and var < 10)
print(var > 3 or var < 4)
print(not (var > 3 and var < 10))
```



```
True
True
False
```

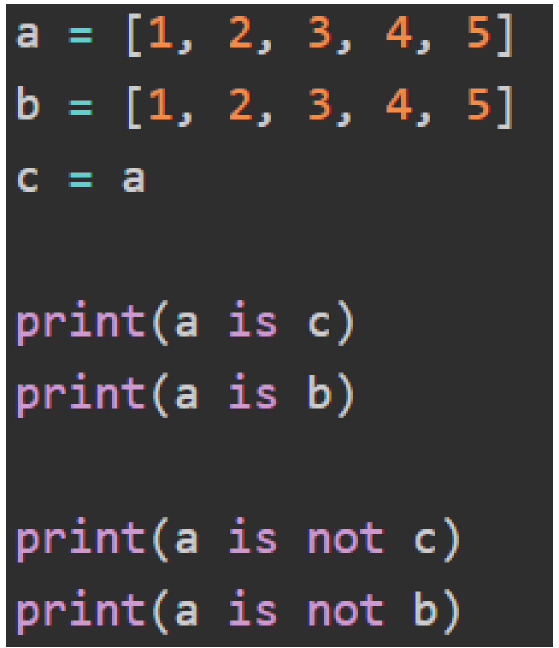
Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below –

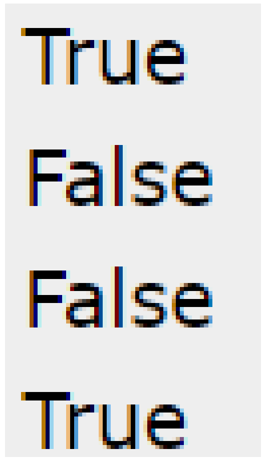


Operator	Description	Example
is	Returns True if both variables are the same object and false otherwise.	a is b
is not	Returns True if both variables are not the same object and false otherwise.	a is not b

Example of Python Identity Operators



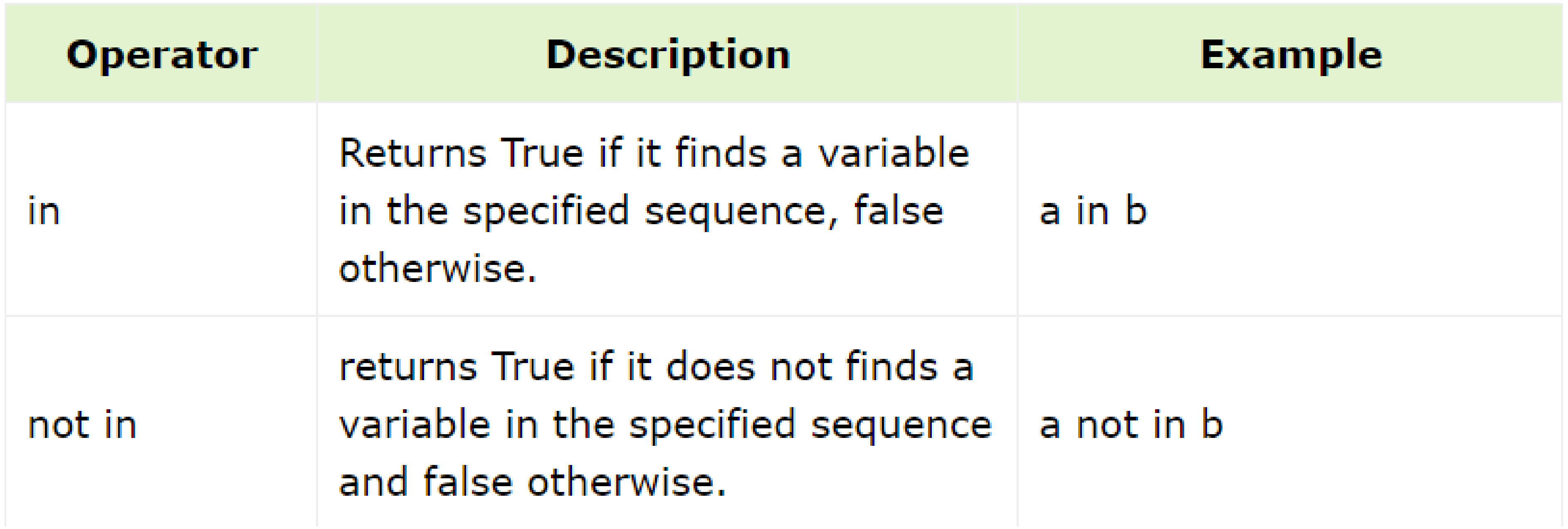
```python
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
c = a

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

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



```
True
False
False
True
```

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below –



Operator	Description	Example
in	Returns True if it finds a variable in the specified sequence, false otherwise.	a in b
not in	returns True if it does not finds a variable in the specified sequence and false otherwise.	a not in b

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Complement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'

7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

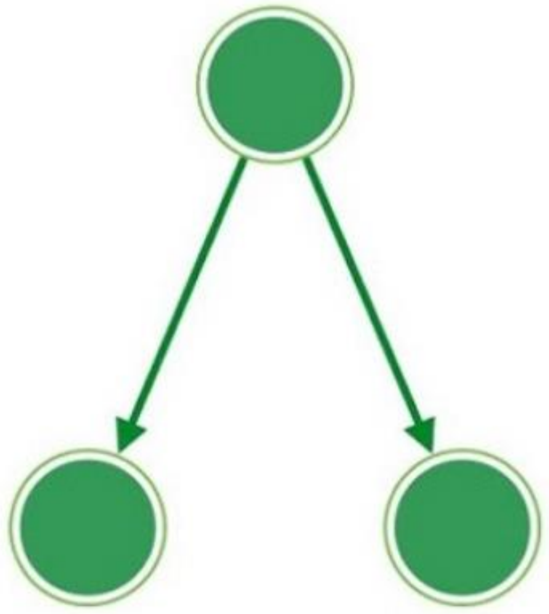
Python - Control Flow

- Python program control flow is regulated by various types of conditional statements, loops, and function calls. By default, the instructions in a computer program are executed in a sequential manner, from top to bottom, or from start to end.
- However, such sequentially executing programs can perform only simplistic tasks. We would like the program to have a decision-making ability, so that it performs different steps depending on different conditions.
- Most programming languages including Python provide functionality to control the flow of execution of instructions.
- Normally, there are two type of control flow statements in any programming language and Python also supports them.

Decision Making Statements

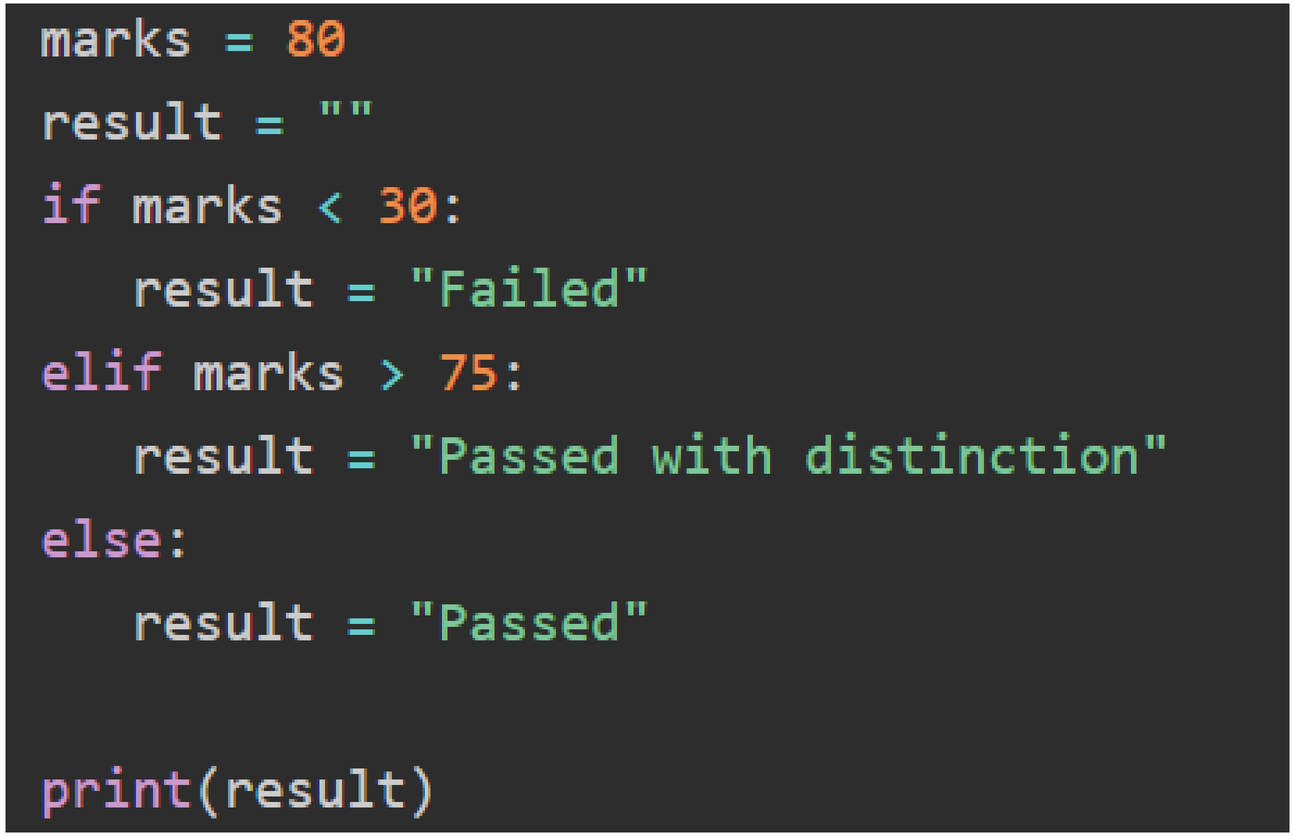
Decision making statements are used in the Python programs to make them able to decide which of the alternative group of instructions to be executed, depending on value of a certain Boolean expression.

The following diagram illustrates how decision-making statements work −



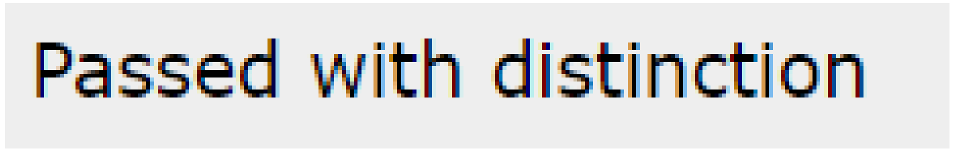
if Statements

Python provides if..elif..else control statements as a part of decision marking. Following is a simple example which makes use of if..elif..else. You can try to run this program using different marks and verify the result.



```python
marks = 80
result = ""
if marks < 30:
   result = "Failed"
elif marks > 75:
   result = "Passed with distinction"
else:
   result = "Passed"

print(result)
```

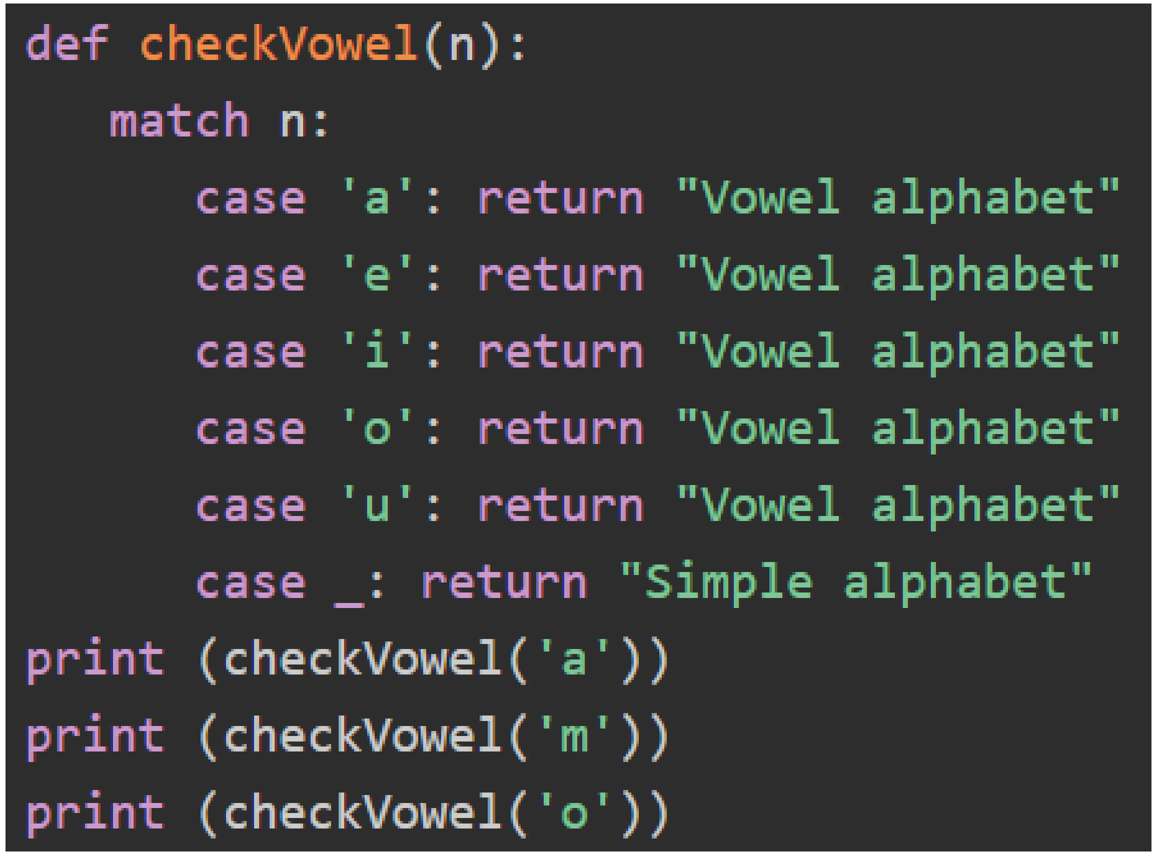
Result:



```
Passed with distinction
```

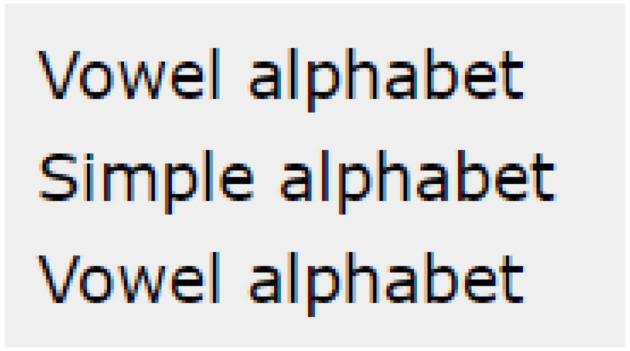
match Statement

Python supports Match-Case statement, which can also be used as a part of decision making. Following is a simple example which makes use of match statement.



```
def checkVowel(n):
   match n:
      case 'a': return "Vowel alphabet"
      case 'e': return "Vowel alphabet"
      case 'i': return "Vowel alphabet"
      case 'o': return "Vowel alphabet"
      case 'u': return "Vowel alphabet"
      case _: return "Simple alphabet"
print (checkVowel('a'))
print (checkVowel('m'))
print (checkVowel('o'))
```

Result:

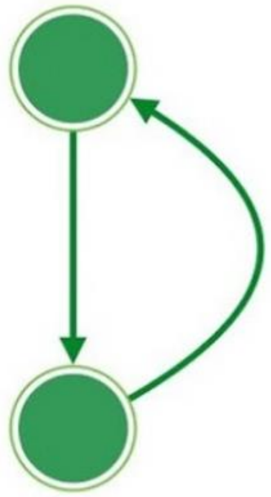


```
Vowel alphabet
Simple alphabet
Vowel alphabet
```

Loops or Iteration Statements

Most of the processes require a group of instructions to be repeatedly executed. In programming terminology, it is called a loop. Instead of the next step, if the flow is redirected towards any earlier step, it constitutes a loop.

The following diagram illustrates how the looping works −



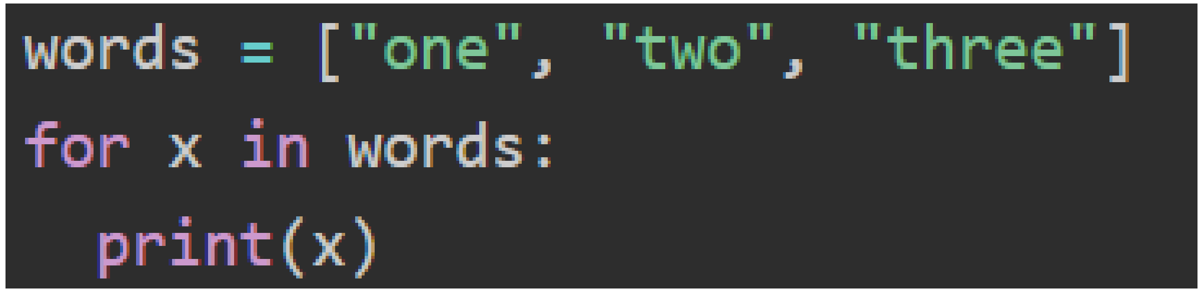
If the control goes back unconditionally, it forms an infinite loop which is not desired as the rest of the code would never get executed.

In a conditional loop, the repeated iteration of block of statements goes on till a certain condition is met. Python supports a number of loops like for loop, while loop which we will study in next chapters.

for Loop

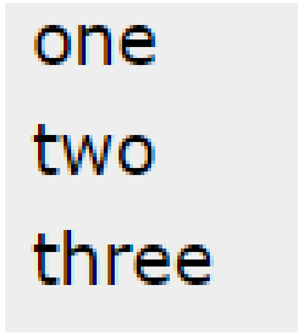
The for loop iterates over the items of any sequence, such as a list, tuple or a string.

Following is an example which makes use of For Loop to iterate through an array in Python:



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

This will produce following result:

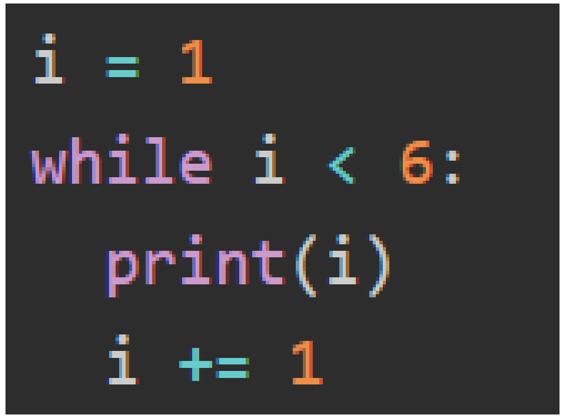


```
one
two
three
```

while Loop

The while loop repeatedly executes a target statement as long as a given boolean expression is true.

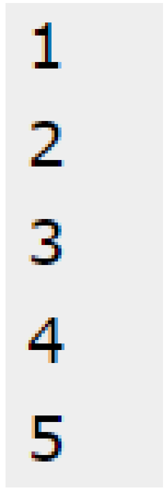
Following is an example which makes use of While Loop to print first 5 numbers in Python:



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

This will produce following result:

Result:



```
1
2
3
4
5
```

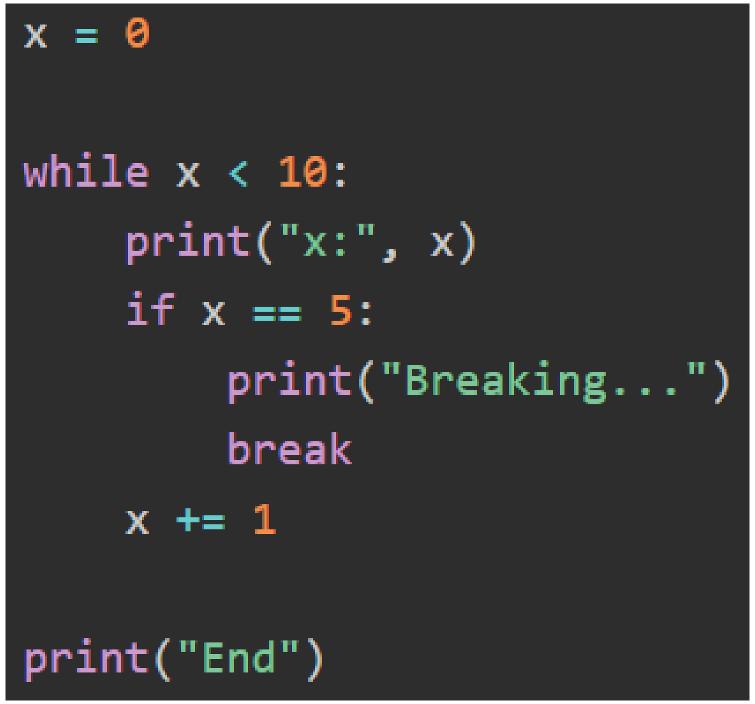
Jump Statements

The jump statements are used to jump on a specific statement by breaking the current flow of the program. In Python, there are two jump statements break and continue.

The break Statement

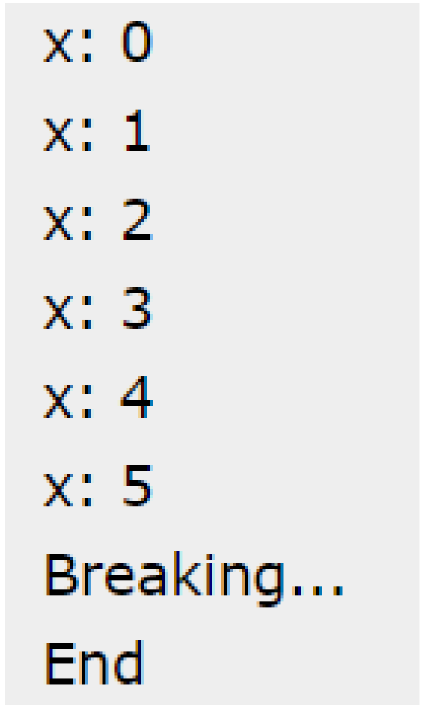
It terminates the current loop and resumes execution at the next statement.

The following example demonstrates the use of break statement -



```python
x = 0

while x < 10:
    print("x:", x)
    if x == 5:
        print("Breaking...")
        break
    x += 1

print("End")
```

Result:

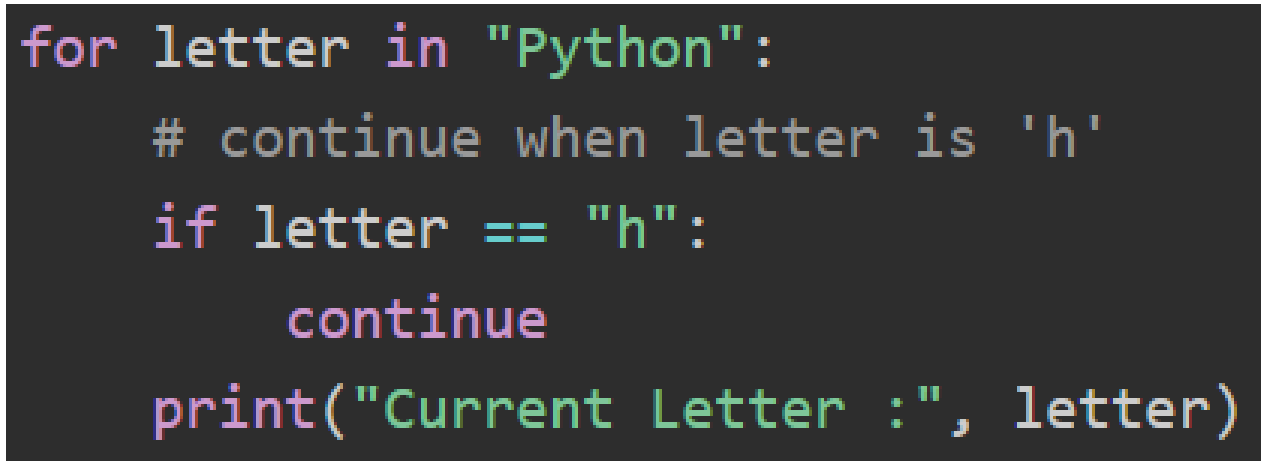


```
x: 0
x: 1
x: 2
x: 3
x: 4
x: 5
Breaking...
End
```

continue Statement

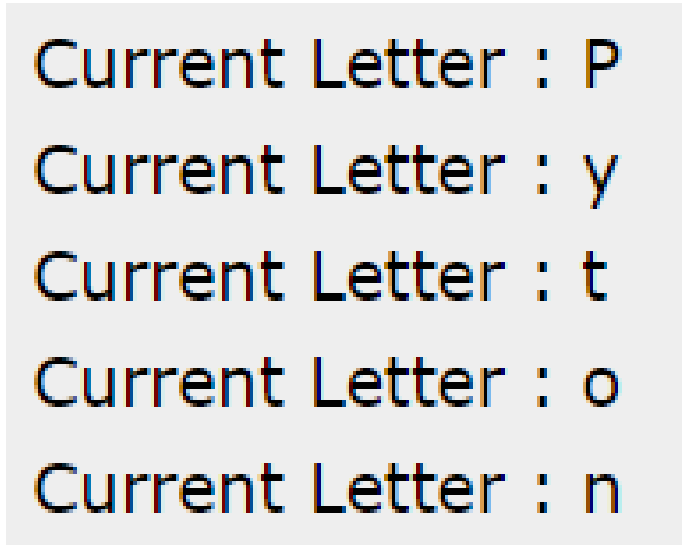
It skips the execution of the program block and returns the control to the beginning of the current loop to start the next iteration.

The following example demonstrates the use of continue statement -



```python
for letter in "Python":
    # continue when letter is 'h'
    if letter == "h":
        continue
    print("Current Letter :", letter)
```

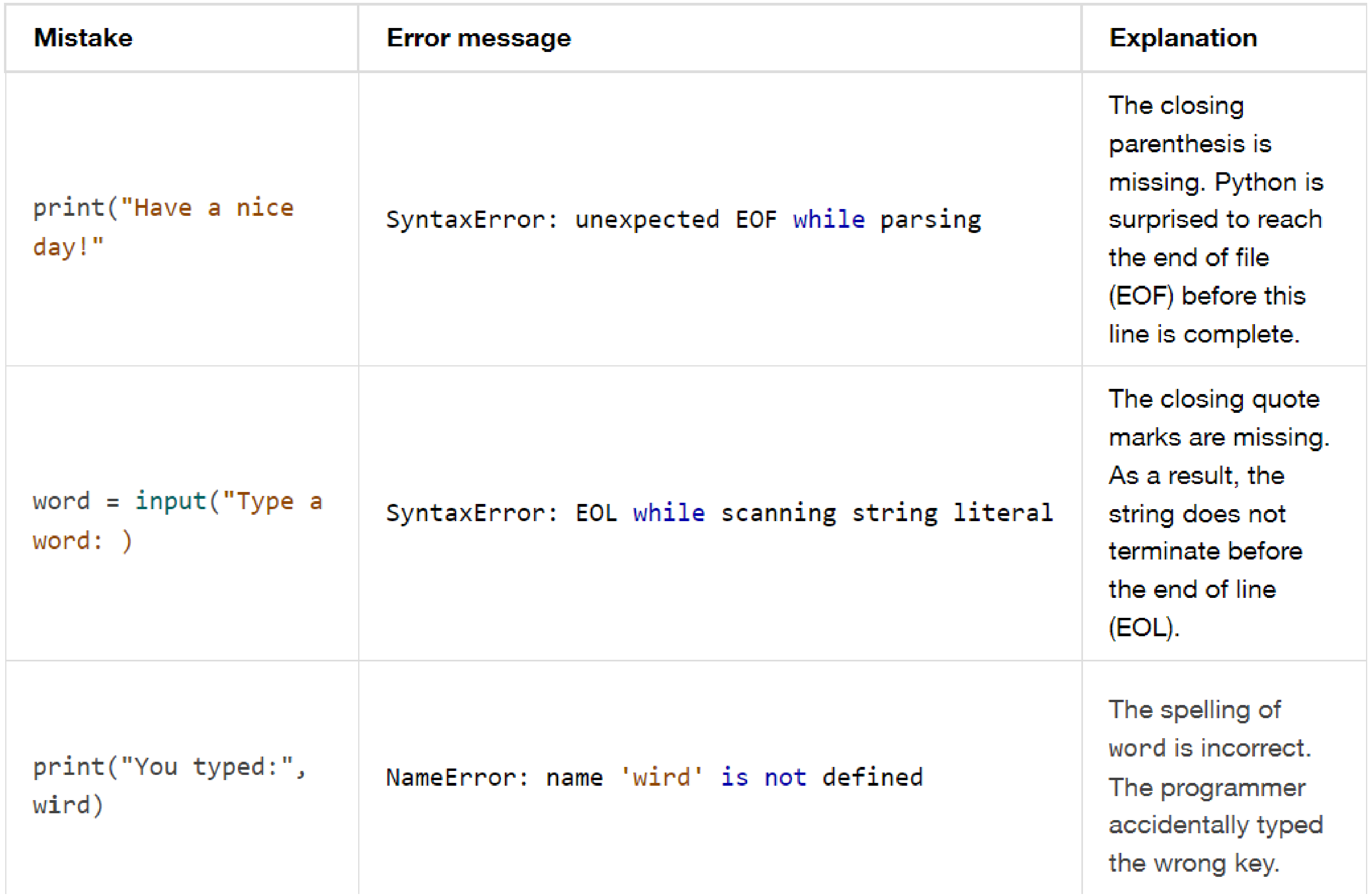
Result:



```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

Common types of errors

Different types of errors may occur when running Python programs. When an error occurs, knowing the type of error gives insight about how to correct the error. The following table shows examples of mistakes that anyone could make when programming.



Mistake	Error message	Explanation
`print("Have a nice day!"`	`SyntaxError: unexpected EOF while parsing`	The closing parenthesis is missing. Python is surprised to reach the end of file (EOF) before this line is complete.
`word = input("Type a word: )`	`SyntaxError: EOL while scanning string literal`	The closing quote marks are missing. As a result, the string does not terminate before the end of line (EOL).
`print("You typed:", wird)`	`NameError: name 'wird' is not defined`	The spelling of word is incorrect. The programmer accidentally typed the wrong key.

`prints("You typed:", word)`	`NameError: name 'prints' is not defined`	The spelling of `print` is incorrect. The programmer accidentally typed an extra letter.
` print("Hello")`	`IndentationError: unexpected indent`	The programmer accidentally typed a space at the start of the line.
`    print("Goodbye")`	`IndentationError: unexpected indent`	The programmer accidentally pressed the Tab key at the start of the line.

Which program stores and retrieves a variable correctly?

a. print("Total =", total)

  total = 6

b. total = 6

  print("Total =", total)

c. print("Total =", total)

  total = input()

Which is the assignment operator?

a. :

b. ==

c. =

Which is a valid assignment?

a. temperature = 98.5

b. 98.5 = temperature

c. temperature - 23.2a.

Which can be used as a variable name?

a. median

b. class

c. import

Why is the name, 2nd_input, not a valid variable name?

a. contains an underscore

b. starts with a digit

c. is a keyword

Which would be a good name for a variable storing a zip code?

a. z

b. var_2

c. zip_code

Given the variable name, DogBreed, which improvement conforms to Python's style guide?

a. dog_breed

b. dogBreed

c. dog-breed

Which of the following is a string?

a. Hello!

b. 29

c. "7 days"

Which line of code assigns a string to the variable email?

a. "fred78@gmail.com"

b. "email = fred78@gmail.com"

c. email = "fred78@gmail.com"

Which is a valid string?

a. I know you'll answer correctly!

b. 'I know you'll answer correctly!'

c. "I know you'll answer correctly!"

Which is a valid string?

a. You say "Please" to be polite

b. "You say "Please" to be polite"

c. 'You say "Please" to be polite'

What is the return value for len("Hi Ali")?

a. 2

b. 5

c. 6

What is the length of an empty string variable ("")?

a. undefined

b. 0

c. 2

What is the output of the following code?

```
number = "12"
number_of_digits = len(number)
print("Number", number, "has", number_of_digits, "digits.")
```

a. Number 12 has 12 digits.

b. Number 12 has 2 digits.

c. Number 12 has number_of_digits digits.

Which produces the string "10"?

a. 1 + 0

b. "1 + 0"

c. "1" + "0“

Which produces the string "Awake"?

a. "wake" + "A"

b. "A + wake"

c. "A" + "wake“

A user enters "red" after the following line of code.

color = input("What is your favorite color?")

Which produces the output "Your favorite color is red!"?

a. print("Your favorite color is " + color + !)

b. print("Your favorite color is " + "color" + "!")

c. print("Your favorite color is " + color + "!")

Which of the following assigns "one-sided" to the variable holiday?

a. holiday = "one" + "sided"

b. holiday = one-sided

c. holiday = "one-" + "sided"

What is the value of 4 * 3 ** 2 + 1?

a. 37

b. 40

c. 145

Which part of (1 + 3) ** 2 / 4 evaluates first?

a. 4 ** 2

b. 1 + 3

c. 2 / 4

What is the value of -4 ** 2?

a. -16

b. 16

How many operators are in the following statement?

```
result = -2 ** 3
```

a. 1

b. 2

c. 3

For each program below, what type of error will occur?

```
print("Breakfast options:")
 print("A. Cereal")
 print("B. Eggs")
 print("C. Yogurt")
choice = input("What would you like? ")
```

a. IndentationError
b. NameError
c. SyntaxError

```
birth = input("Enter your birth date: )
print("Happy birthday on ", birth)
```

a. IndentationError
b. NameError
c. SyntaxError

```
print("Breakfast options:")
print(" A. Cereal")
print(" B. Eggs")
print(" C. Yogurt")
choice = intput("What would you like? ")
```

a. IndentationError
b. NameError
c. SyntaxError

Thank you!!