What is Python?

Python is a **general purpose**, **interpreted**, **interactive** and **object-oriented scripting** language. It has fewer ***syntactical constructions*** than other languages. We don't need to use **data types** to declare **variables** because it is ***dynamically typed.***

- **Python is interpreted:** Python is processed at **runtime** by the ***interpreter***. You do not need to ***compile*** your program before executing it.
- **Python is Interactive:** You can actually sit at a ***Python prompt*** and **interact** with the ***interpreter*** directly to write your programs.
- **Python is Object-Oriented:** Python supports ***Object-Oriented*** style or technique of programming that ***encapsulates*** code within objects.

History of Python:

Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

- The implementation of Python was started in the December 1989 by Guido Van Rossum at CWI in Netherland.
- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).
- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.
- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one – and preferably only one -- obvious way to do it." Python 3.8.5 is the latest version of Python 3.

Why Python? Because of its Python Features:

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined readable syntax which allows to pick up language quickly.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Interpreted Language:** Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.
- **Cross-platform Language:** Python can run equally on different platforms such as Windows, Linux, UNIX and Macintosh etc. So, we can say that Python is a portable language.
- **Databases:** Python provides interfaces to all major commercial databases.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.
- **Large Standard Library:** Python has a large and broad library and provides rich set of module and functions for rapid application development.
- **Object-Oriented Language:** Python supports object oriented language and concepts of classes and objects come into existence.

First Python Program?

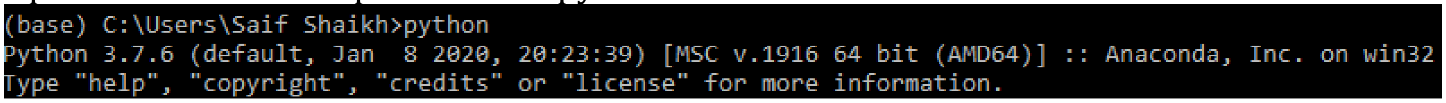
Python provides us the two ways to run a program:

- Using Interactive interpreter prompt
- Using a script file

1) Interactive interpreter prompt:

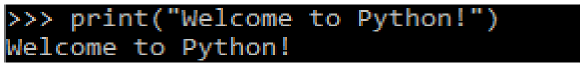
Python provides us the feature to execute the python statement one by one at the interactive prompt with print ( ) function by passing string message into this function.

Open Command Prompt and enter python:



```
(base) C:\Users\Saif Shaikh>python
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
```

Enter Text:



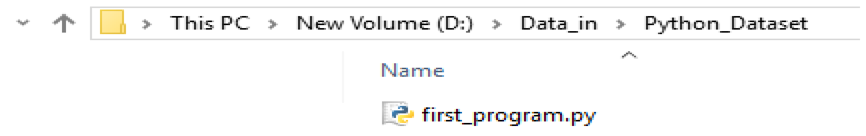
```
>>> print("Welcome to Python!")
Welcome to Python!
```

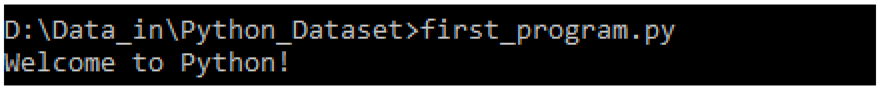
2) Using a script file:

Interpreter prompt is good to run the individual statements of the code. However, we cannot write the code every-time on the terminal. We need to write our code into a file which can be executed later.

For this purpose, open an editor like notepad, create a file named first.py (python used .py extension) and write the following code in it.

Folder: D:\Data_in\Python_Dataset

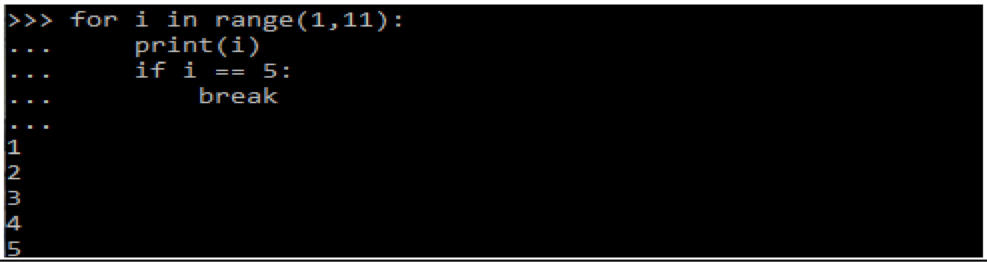




```
D:\Data_in\Python_Dataset>first_program.py
Welcome to Python!
```

Python Indentation:

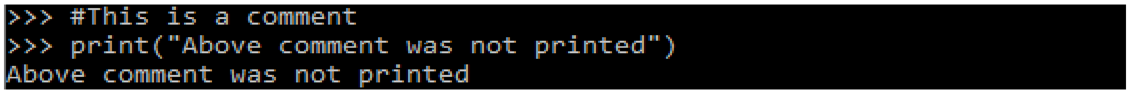
A code block (body of a function, loop, etc.) starts with indentation and ends with the first un-indented line. The amount of indentation is up to you, but it must be consistent throughout that block. Generally, four whitespaces are used for indentation and are preferred over tabs.



```
>>> for i in range(1,11):
...     print(i)
...     if i == 5:
...         break
...
1
2
3
4
5
```

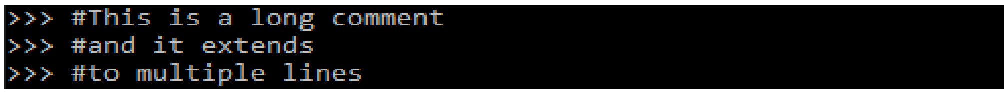
Comments:
Comments can be used to explain Python code & it makes the code more readable.

Single Line Comment:
In Python, we use the **hash (#)** symbol to start writing a comment. It extends up to the newline character.

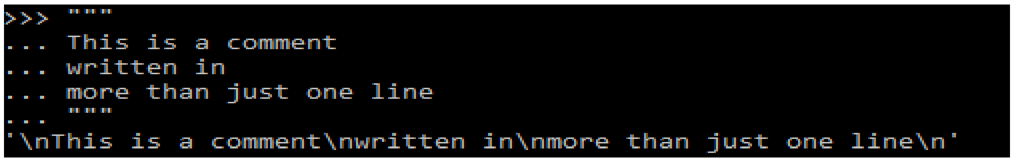


```
>>> #This is a comment
>>> print("Above comment was not printed")
Above comment was not printed
```

Multi-line Comment:
We can have comments that extend up to multiple lines. One way is to use the **hash (#)** symbol at the beginning of each line. Another way of doing this is to use triple quotes, either **''' ''' OR """ """**. **Multi lined comment** can be given inside **triple** quotes.



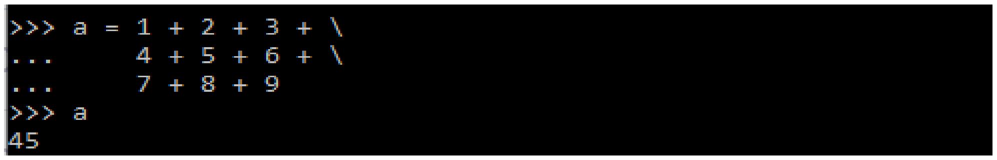
```
>>> #This is a long comment
>>> #and it extends
>>> #to multiple lines
```



```
>>> """
... This is a comment
... written in
... more than just one line
... """
'\nThis is a comment\nwritten in\nmore than just one line\n'
```

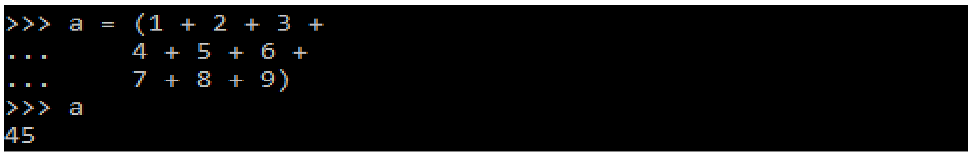
Python Statement:
Instructions that a Python **interpreter** can **execute** are called **statements**. For example, a = 1 is an assignment statement.

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



```
>>> a = 1 + 2 + 3 + \
...     4 + 5 + 6 + \
...     7 + 8 + 9
>>> a
45
```

This is an **explicit** line continuation. In Python, **line continuation** is implied inside **parentheses ( ), brackets [ ]**, and **braces { }.** For instance, we can implement the above **multi-line** statement as:



```
>>> a = (1 + 2 + 3 +
...     4 + 5 + 6 +
...     7 + 8 + 9)
>>> a
45
```

Python Identifiers:

A Python **identifier** is a name used to **identify** a **variable**, **function**, **class**, **module** or other Object. An **identifier** starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python **does not** allow punctuation characters such as @, $, and % within identifiers. Python is a ***case sensitive*** programming language. Thus, **Saif** and **saif** are two different identifiers in Python.

Here are naming conventions for Python identifiers:

- **Class** names start with an **UPPERCASE** letter. All other identifiers start with a **LOWERCASE** letter.
- Starting an identifier with a **single leading underscore** indicates that the identifier is **PRIVATE.**
- Starting an identifier with **two leading underscores** indicates a **STRONG PRIVATE** identifier.
- If the identifier also ends with **two trailing underscores**, the identifier is a **language defined special name**.
- The first character of the **variable** must be an alphabet or underscore (_).
- All the characters **except** the first character may be an alphabet of lower-case (a-z), upper-case (A-Z), underscore or digit (0-9).
- Identifier name must **not** contain any white-space, or special character (!, @, #, %, ^, &, *).
- Identifier name must **not** be similar to any **keyword defined** in the language.
- Identifier names are **case sensitive** e.g. my name, and MyName is **not** the same.
- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

Reserved Words:

The following list shows the Python keywords. These are ***reserved words*** and you cannot Use them as **constants** or **variables** or any other **identifier** names. All the Python keywords contain **lowercase letters** only.

and	exec	not	as	finally	or
assert	for	pass	break	from	print
class	global	raise	continue	if	return
def	import	try	del	in	while
elif	is	with	else	lambda	yield
except					

Quotation in Python:

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string. The triple quotes are used to span the string across multiple lines. For example, all the following are legal:

Word = 'word'
Sentence = "This is a sentence."
Paragraph = """This is a paragraph.
It is made up of multiple lines and sentences."""

Multiple Statements on a Single Line:

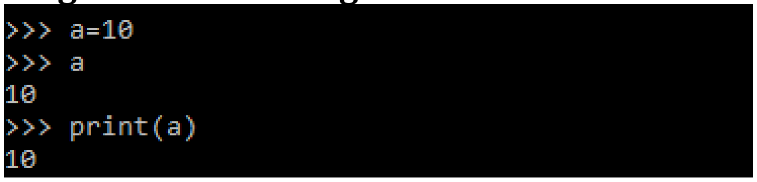
The **semicolon (;)** allows **multiple statements** on a **single** line given that no statement starts a new code block. Here is a sample snip using the semicolon:

```
import sys; x = 'saif'; sys.stdout.write (x + '\n')
```

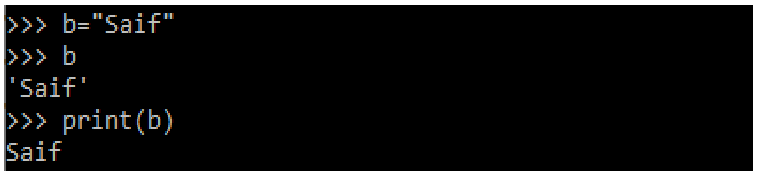
Python Variables:

- **Variable** is a name which is used to refer **memory location** and they are also known as **identifier** to **hold values**.
- In Python, we **don't** need to specify the **type** of variable because Python is a **type infer language** and smart enough to get variable type.
- Variable names can be a group of both letters and digits, but they have to begin with a **letter** or an **underscore**.
- It is recommended to use **lowercase** letters for **variable** name. **Saif** and **saif** both are two **different** variables.

Single Variable Assignment:

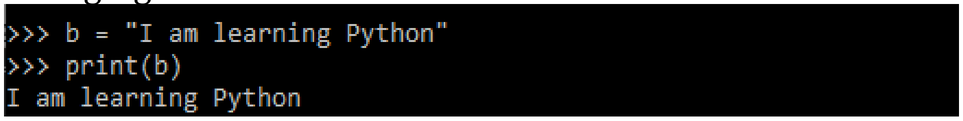


```
>>> a=10
>>> a
10
>>> print(a)
10
```



```
>>> b="Saif"
>>> b
'Saif'
>>> print(b)
Saif
```

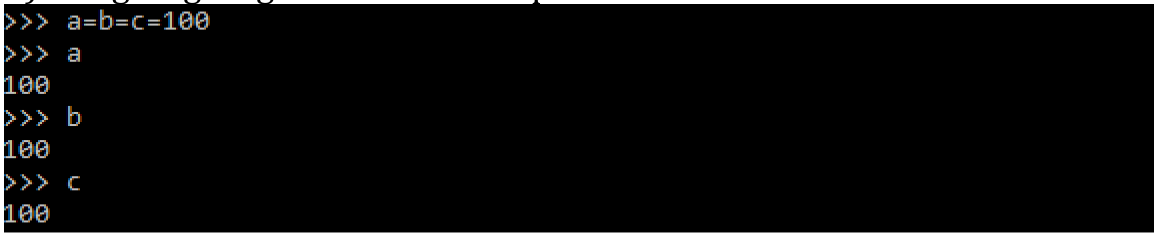
Changing Values in Variables:



```
>>> b = "I am learning Python"
>>> print(b)
I am learning Python
```

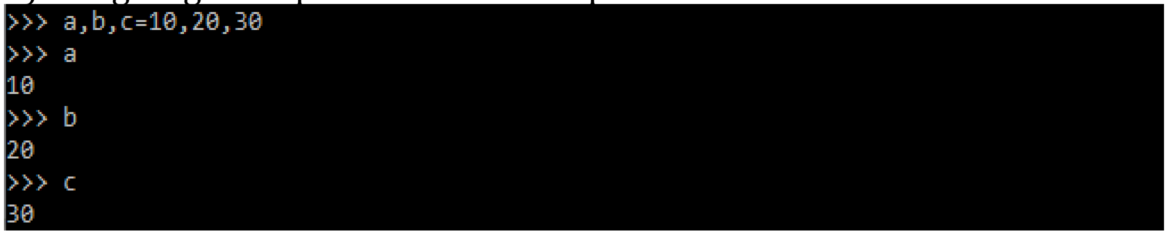
Multiple Variable Assignment:

1) Assigning single value to multiple variables



```
>>> a=b=c=100
>>> a
100
>>> b
100
>>> c
100
```

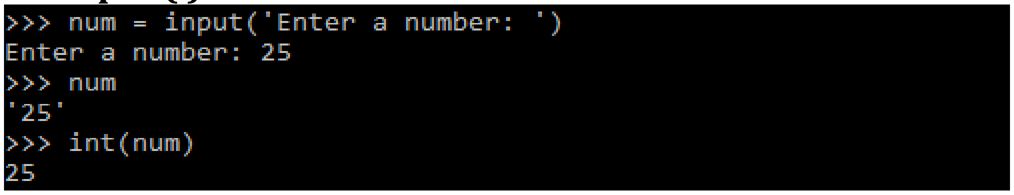
2) Assigning multiple values to multiple variables:



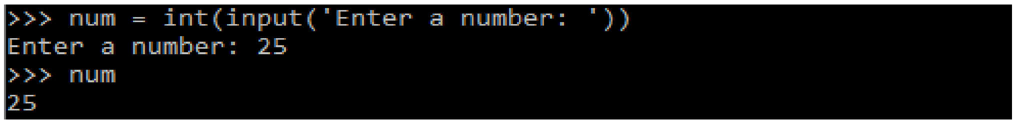
```
>>> a,b,c=10,20,30
>>> a
10
>>> b
20
>>> c
30
```

Input Variables:

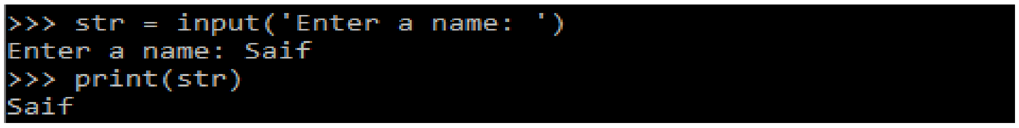
To allow flexibility, we might want to take the **input** from the user. In Python, we have the **input ( )** function to allow this.



```
>>> num = input('Enter a number: ')
Enter a number: 25
>>> num
'25'
>>> int(num)
25
```



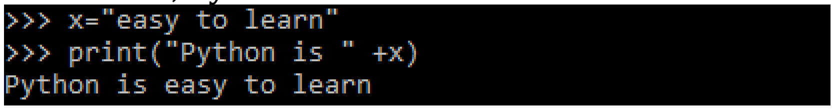
```
>>> num = int(input('Enter a number: '))
Enter a number: 25
>>> num
25
```



```
>>> str = input('Enter a name: ')
Enter a name: Saif
>>> print(str)
Saif
```

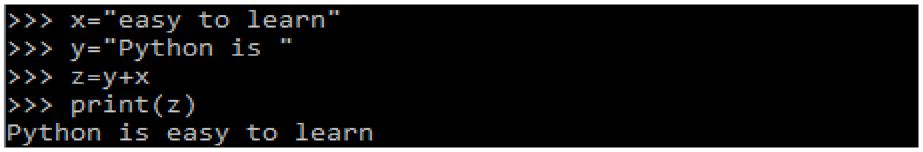
Output Variables

The Python **print** statement is often used to **output** variables. To **combine** both **text** and a **variable**, Python uses the **+ character**:



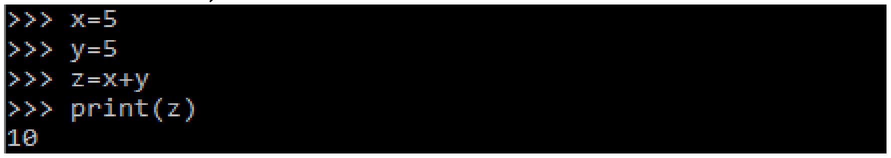
```
>>> x="easy to learn"
>>> print("Python is " +x)
Python is easy to learn
```

You can also use the **+ character** to **add** a variable to another variable:



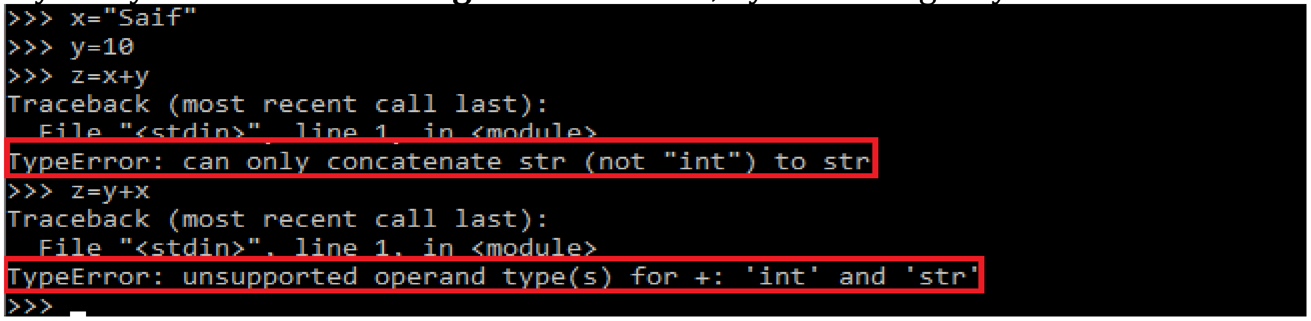
```
>>> x="easy to learn"
>>> y="Python is "
>>> z=y+x
>>> print(z)
Python is easy to learn
```

For numbers, the **+ character** works as a **mathematical** operator:



```
>>> x=5
>>> y=5
>>> z=x+y
>>> print(z)
10
```

If you try to combine a **string** and a **number**, Python will give you an **error**:

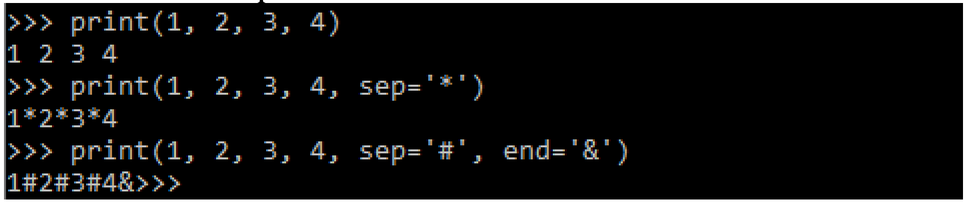


```
>>> x="Saif"
>>> y=10
>>> z=x+y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> z=y+x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> _
```

print ( ):
Syntax: print (*objects, sep = ' ', end = '\n', file = sys.stdout, flush = False)

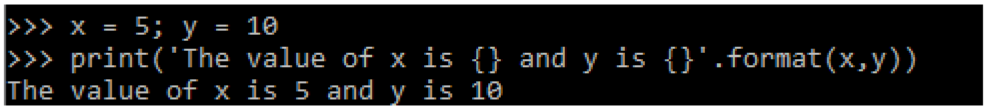
Here, **objects** is the value(s) to be printed.
The **sep** separator is used between the values. It defaults into a **space** character.
After all values are printed, **end** is printed. It defaults into a **new** line.
The **file** is the object where the values are printed and its default value is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

Here is an example to illustrate this.



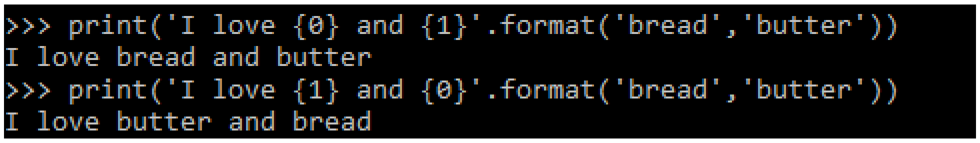
```
>>> print(1, 2, 3, 4)
1 2 3 4
>>> print(1, 2, 3, 4, sep='*')
1*2*3*4
>>> print(1, 2, 3, 4, sep='#', end='&')
1#2#3#4&>>>
```

Output formatting:
Sometimes we would like to **format** our **output** to make it look **attractive**. This can be done by using the **str.format ( ) method**. This **method** is **visible** to any **string** object.



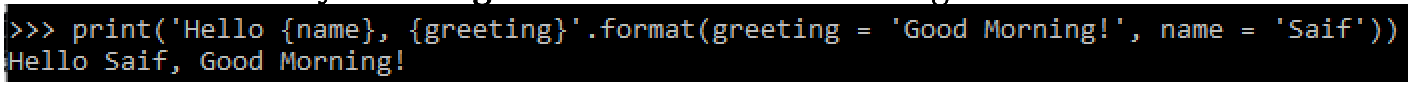
```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is {}'.format(x,y))
The value of x is 5 and y is 10
```

Here, the **curly braces { }** are used as **placeholders**. We can specify the **order** in which they are **printed** by using **numbers index**.



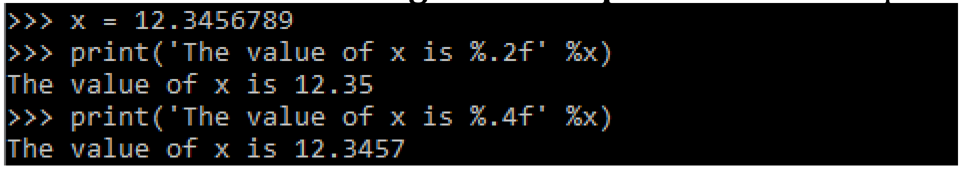
```
>>> print('I love {0} and {1}'.format('bread','butter'))
I love bread and butter
>>> print('I love {1} and {0}'.format('bread','butter'))
I love butter and bread
```

We can even use **keyword arguments** to **format** the string.



```
>>> print('Hello {name}, {greeting}'.format(greeting = 'Good Morning!', name = 'Saif'))
Hello Saif, Good Morning!
```

We can also **format strings** with **% operator** to accomplish this.

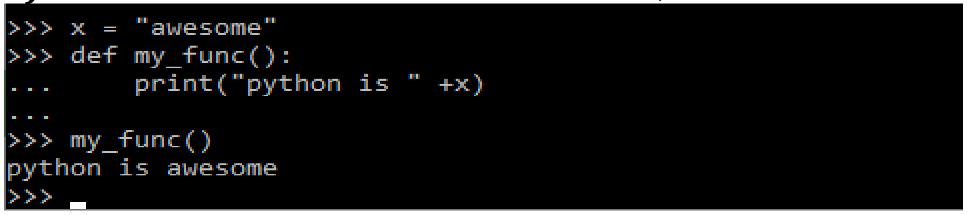


```
>>> x = 12.3456789
>>> print('The value of x is %.2f' %x)
The value of x is 12.35
>>> print('The value of x is %.4f' %x)
The value of x is 12.3457
```

Global Variables

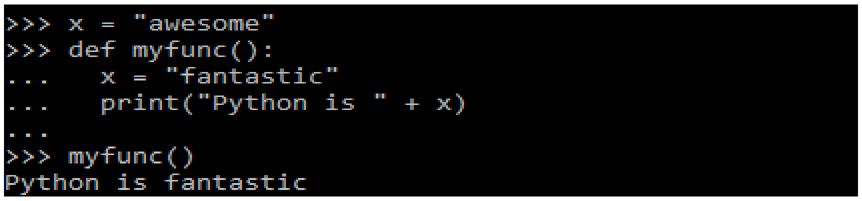
- **Variables** that are created **outside** of a **function** are known as **global variables.**
- **Global variables** can be used by **everyone**, both **inside** of functions and **outside.**

a) Create a variable **outside** of a function, and use it **inside** the function.



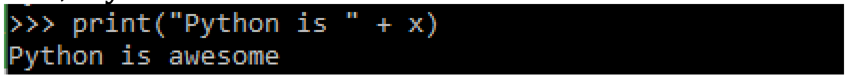
```
>>> x = "awesome"
>>> def my_func():
...     print("python is " +x)
...
>>> my_func()
python is awesome
>>> _
```

b) If you create a **variable** with the **same** name **inside** a function, this variable will be **local**, and can only be used **inside** the function. The **global variable** with the **same** name will remain as it was **global** and with the **original** value.



```
>>> x = "awesome"
>>> def myfunc():
...    x = "fantastic"
...    print("Python is " + x)
...
>>> myfunc()
Python is fantastic
```

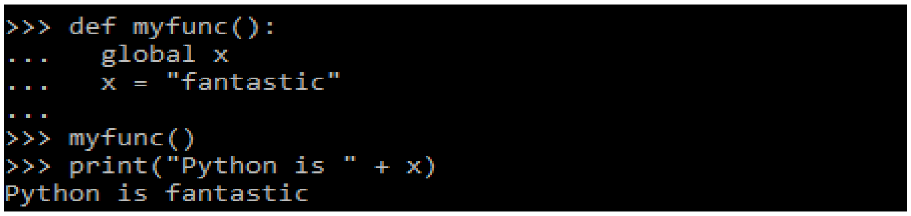
But, if you **call x** it would take **variable outside** a function.



```
>>> print("Python is " + x)
Python is awesome
```

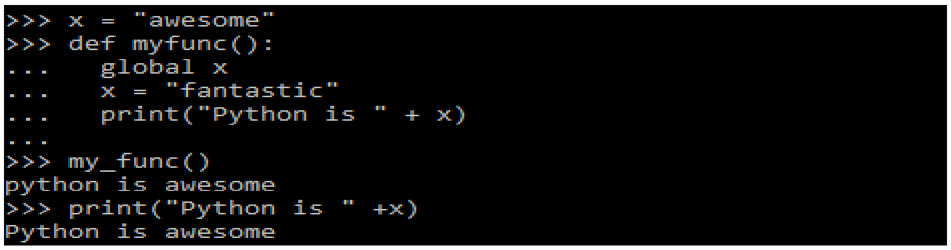
The global Keyword:

Normally, when you create a variable **inside** a function, that **variable** is **local**, and can only be used **inside** that function. To create a **global variable inside** a function, you can use the **global** keyword.



```
>>> def myfunc():
...    global x
...    x = "fantastic"
...
>>> myfunc()
>>> print("Python is " + x)
Python is fantastic
```

Use the **global keyword** if you want to **change** a **variable inside** a function.



```
>>> x = "awesome"
>>> def myfunc():
...    global x
...    x = "fantastic"
...    print("Python is " + x)
...
>>> my_func()
python is awesome
>>> print("Python is " +x)
Python is awesome
```

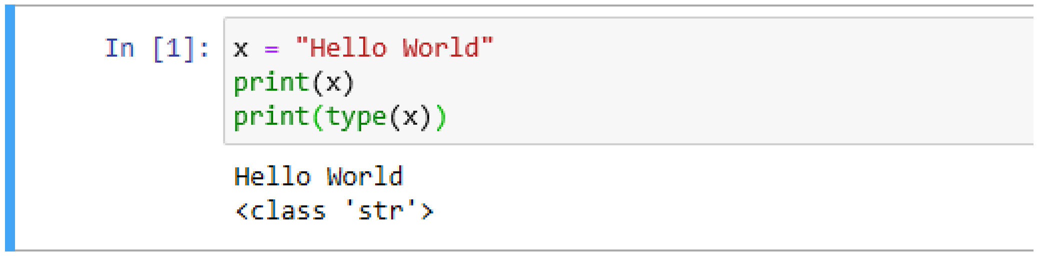
Python Data Types

- **Variables** can hold values of **different** data types.
- Python is a **dynamically typed language** hence we need **not** define the **type** of the **variable** while **declaring** it. The **interpreter implicitly binds** the **value** with its **type**.
- Python provides us the **type ( )** function which **returns** the **type** of the **variable** passed.

Let's understand how to **print** & **cast/set** data types.

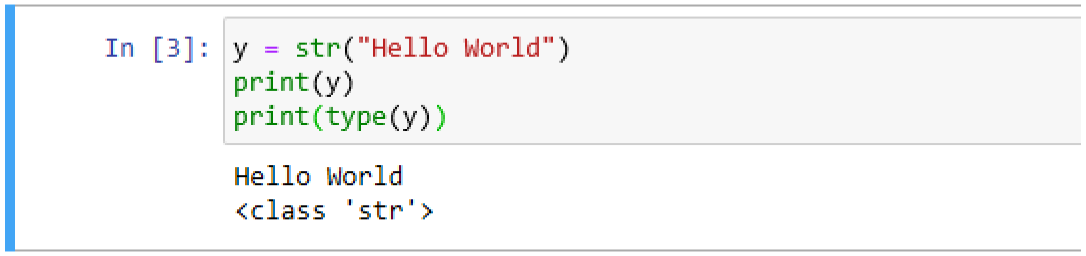
1) String: The string can be defined as the **sequence of characters** represented in the **quotation** marks. In python, we can use **single**, **double**, or **triple** quotes to define a string.

Print Types:



```
In [1]: x = "Hello World"
        print(x)
        print(type(x))

        Hello World
        <class 'str'>
```

Set/Cast Types:

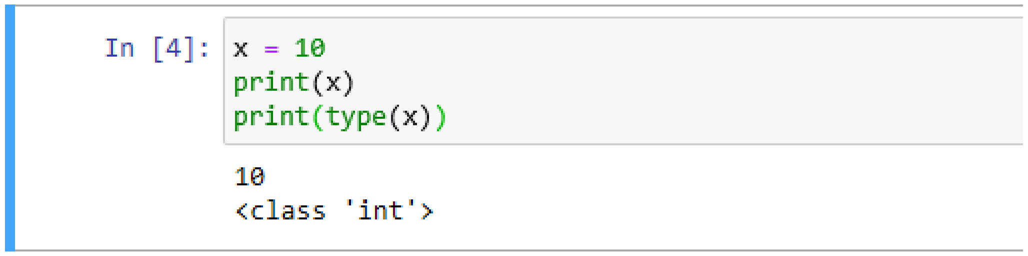


```
In [3]: y = str("Hello World")
        print(y)
        print(type(y))

        Hello World
        <class 'str'>
```

2) Numbers: Number stores **numeric** values. Python creates **number object** when a **number** is assigned to a **variable. Integers**, **floating point** numbers and **complex** numbers fall under Python numbers category.

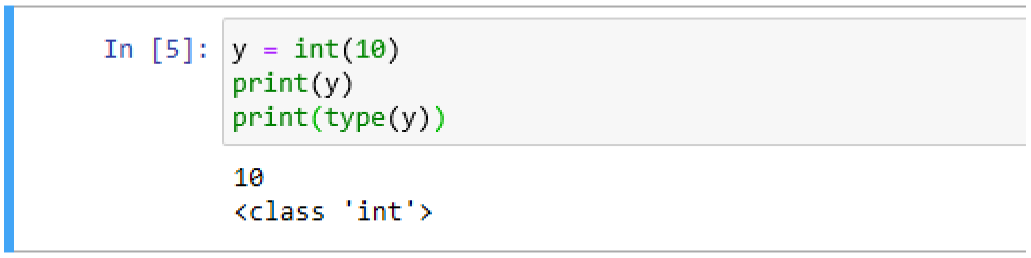
2a) Integer:

Print Types:



```
In [4]: x = 10
        print(x)
        print(type(x))

        10
        <class 'int'>
```

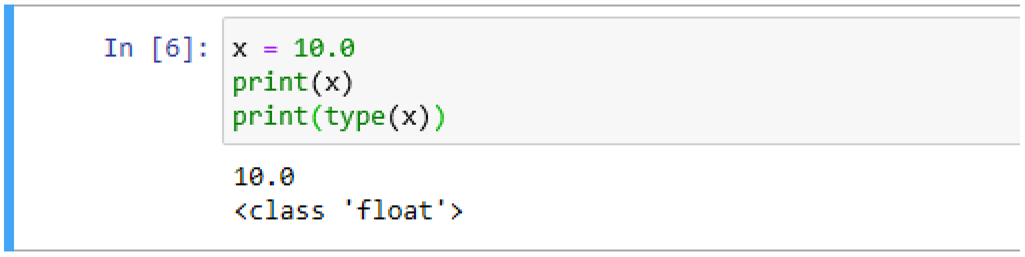
Set/Cast Types:



```
In [5]: y = int(10)
        print(y)
        print(type(y))

10
<class 'int'>
```

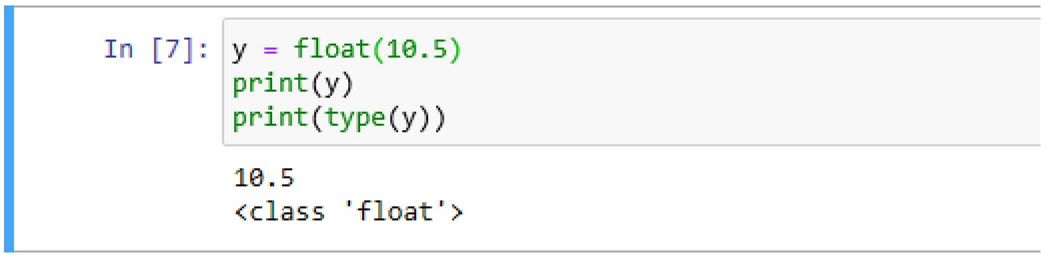
2b) Float:

Print Types:



```
In [6]: x = 10.0
        print(x)
        print(type(x))

10.0
<class 'float'>
```

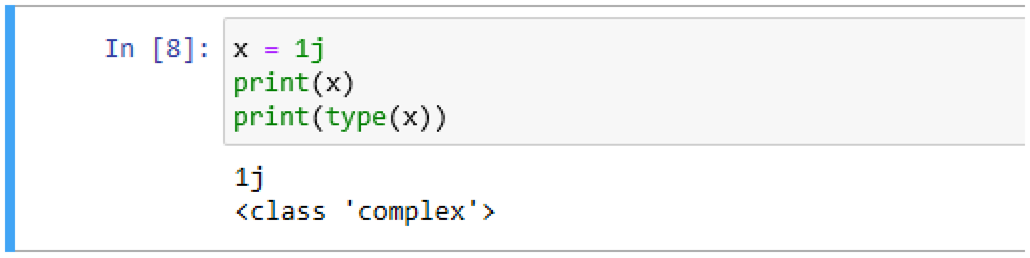
Set/Cast Types:



```
In [7]: y = float(10.5)
        print(y)
        print(type(y))

10.5
<class 'float'>
```

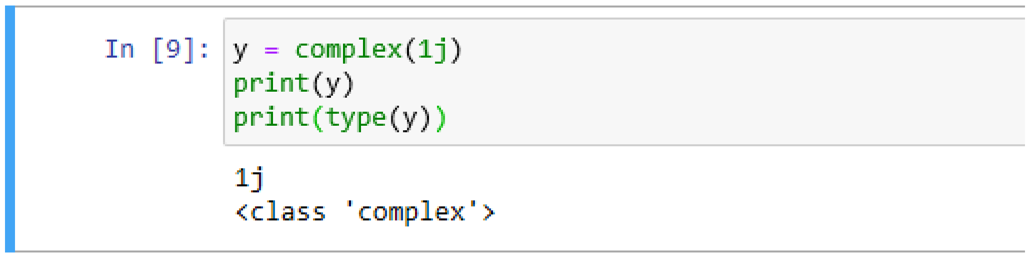
2c) Complex:

Print Types:



```
In [8]: x = 1j
        print(x)
        print(type(x))

1j
<class 'complex'>
```

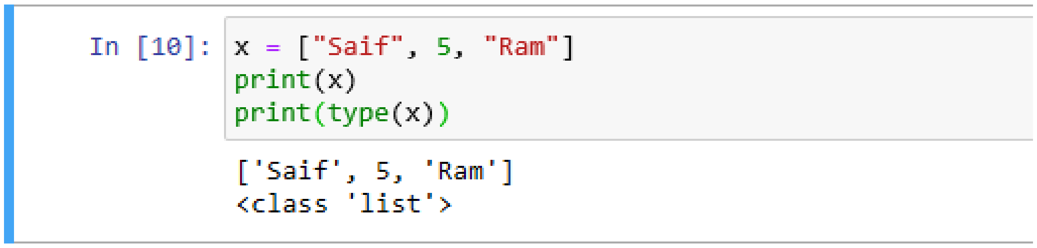
Set/Cast Types:



```
In [9]: y = complex(1j)
        print(y)
        print(type(y))

1j
<class 'complex'>
```

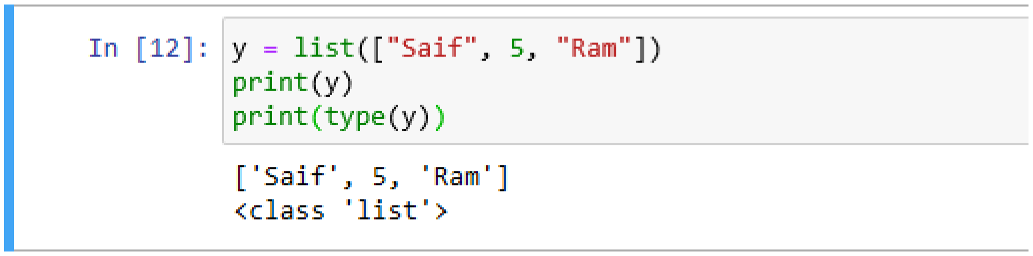
3) List: List is an **ordered sequence** of **items.** All the items in a list do **not** need to be of the **same type**. Items separated by **commas** are enclosed within **brackets [ ]**. We can use the **slicing** operator [ ] to **extract** an item or a **range** of items from a list. The **index** starts from **0** in Python. Lists are **mutable** i.e. the **value of elements** of a list can be **altered**.

Print Types:



```
In [10]: x = ["Saif", 5, "Ram"]
         print(x)
         print(type(x))

         ['Saif', 5, 'Ram']
         <class 'list'>
```

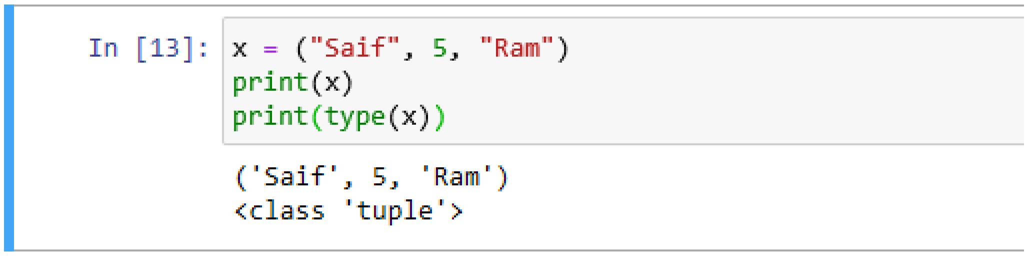
Set/Cast Types:



```
In [12]: y = list(["Saif", 5, "Ram"])
         print(y)
         print(type(y))

         ['Saif', 5, 'Ram']
         <class 'list'>
```

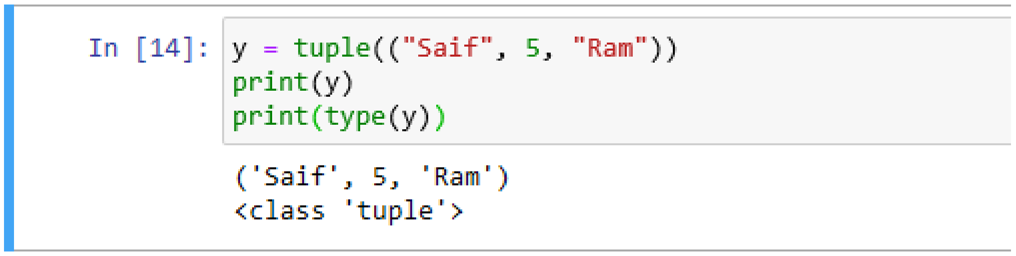
4) Tuple: Tuple is an **ordered sequence** of **items** same as **list**. The only difference is that tuples are **immutable**. Tuples are used to **write-protect** data and are usually **faster** than lists as they **cannot change dynamically**. It is defined within **parentheses ( )** where items are separated by **commas**. We can use the **slicing** operator [ ] to **extract** items but we **cannot** change its **value**.

Print Types:



```
In [13]: x = ("Saif", 5, "Ram")
         print(x)
         print(type(x))

         ('Saif', 5, 'Ram')
         <class 'tuple'>
```

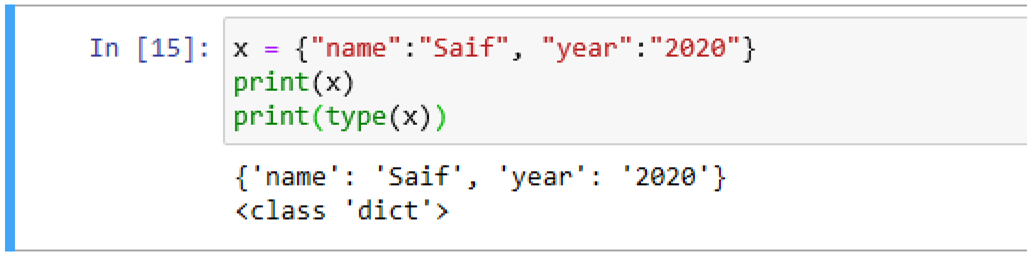
Set/Cast Types:



```
In [14]: y = tuple(("Saif", 5, "Ram"))
         print(y)
         print(type(y))

         ('Saif', 5, 'Ram')
         <class 'tuple'>
```

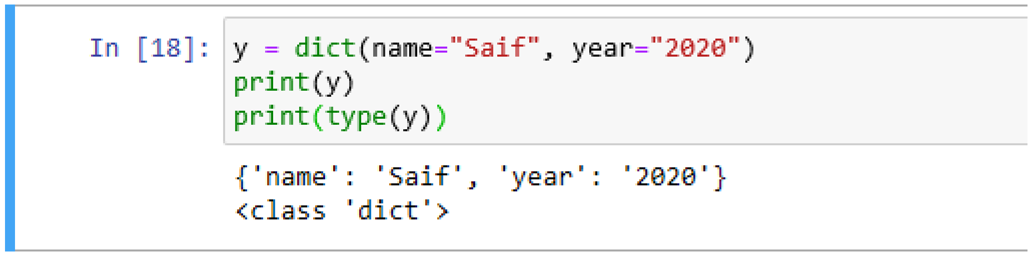
5) Dictionary: Dictionary is an **unordered collection** of **key-value pairs**. In Python, dictionaries are defined within **braces { }** with each item being a **pair** in the form **Key:Value**. Key and value can be of **any** type. We use **key** to retrieve the respective **value**, but **not** the other way around. It is generally used when we have a **huge** amount of data. Dictionaries are **optimized** for retrieving data.

Print Types:



```
In [15]: x = {"name":"Saif", "year":"2020"}
         print(x)
         print(type(x))

         {'name': 'Saif', 'year': '2020'}
         <class 'dict'>
```

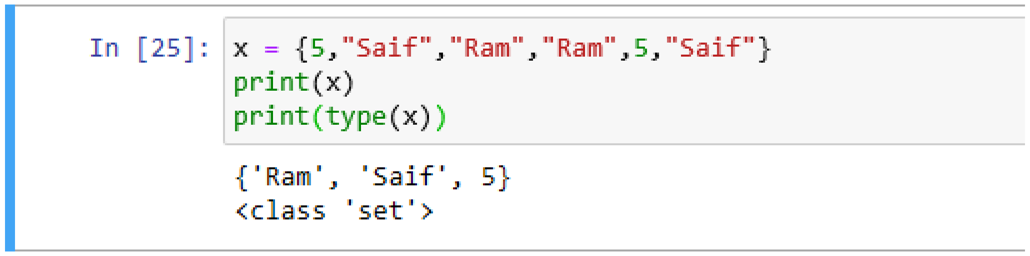
Set/Cast Types:



```
In [18]: y = dict(name="Saif", year="2020")
         print(y)
         print(type(y))

         {'name': 'Saif', 'year': '2020'}
         <class 'dict'>
```

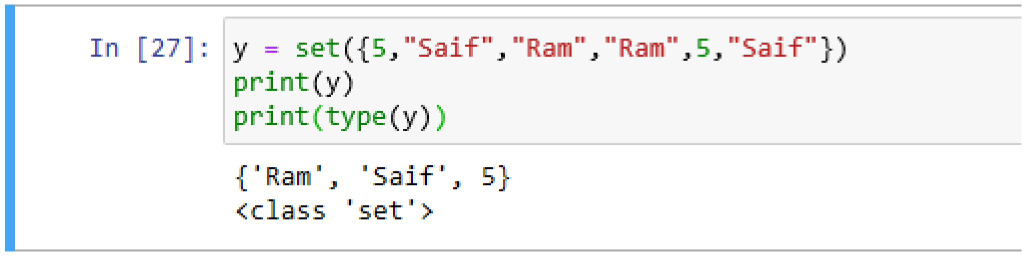
6) Set: Set is an **unordered collection** of **unique items.** Set is defined by values separated by **comma** inside **braces { }**.

Print Types:



```
In [25]: x = {5,"Saif","Ram","Ram",5,"Saif"}
         print(x)
         print(type(x))

         {'Ram', 'Saif', 5}
         <class 'set'>
```

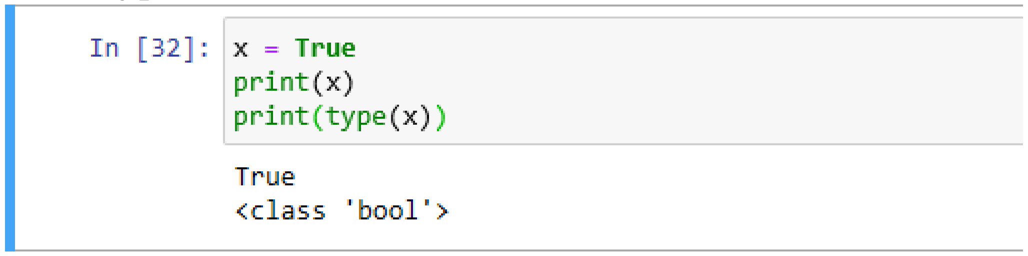
Set/Cast Types:



```
In [27]: y = set({5,"Saif","Ram","Ram",5,"Saif"})
         print(y)
         print(type(y))

         {'Ram', 'Saif', 5}
         <class 'set'>
```

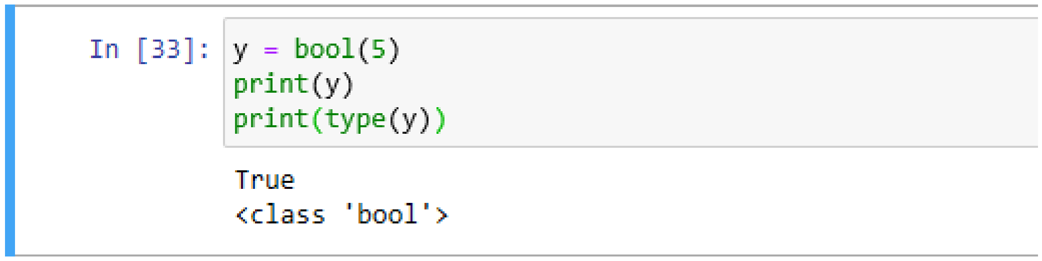
9) Boolean: Booleans represent one of two values: **True** or **False**.

Print Types:

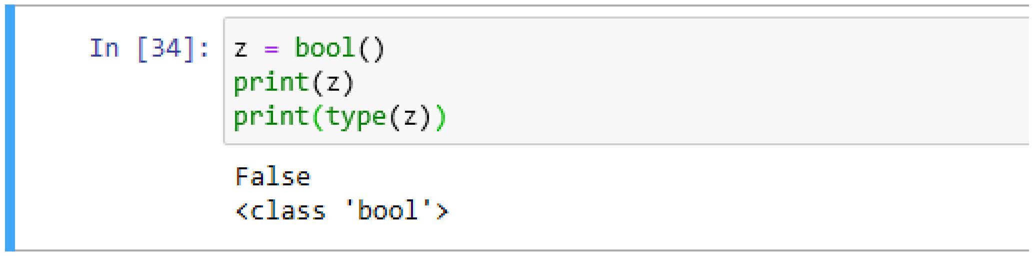


```
In [32]: x = True
         print(x)
         print(type(x))

         True
         <class 'bool'>
```

Set/Cast Types:



```
In [33]: y = bool(5)
         print(y)
         print(type(y))

         True
         <class 'bool'>
```



```
In [34]: z = bool()
         print(z)
         print(type(z))

         False
         <class 'bool'>
```

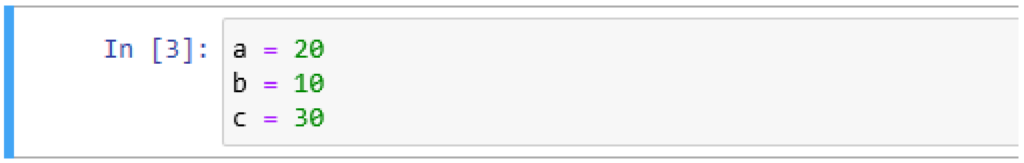
Python Operators:
The operator can be defined as a symbol which is responsible for a particular operation between two operands.

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

1) Arithmetic Operators:
Arithmetic operators are used with **numeric** values to perform **mathematical** operations:

Operator	**Description**
+ (Addition)	It is used to add two operands. **E.g.** if a = 20, b = 10 => a+b = 30
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if a = 20, b = 10 => a - b = 10
/ (Divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2
* (Multiplication)	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (Remainder)	It returns the remainder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
** (Exponent)	It is an exponent operator represented as it calculates the first operand power to second operand.
// (Floor Division)	It gives the floor value of the quotient produced by dividing the two operands.

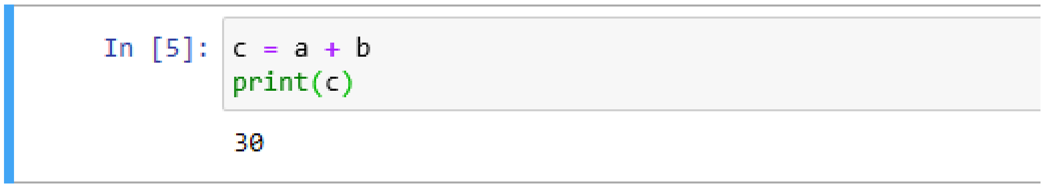
E.g. Below variables assigned:



```
In [3]: a = 20
        b = 10
        c = 30
```

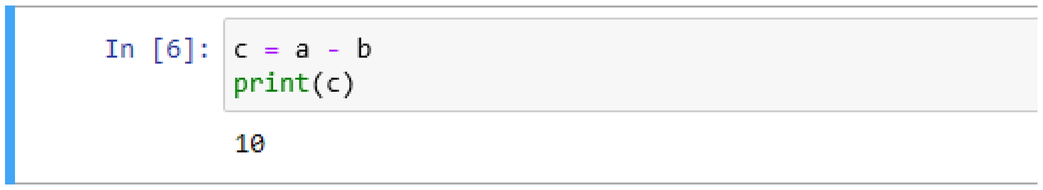
'

Addition:



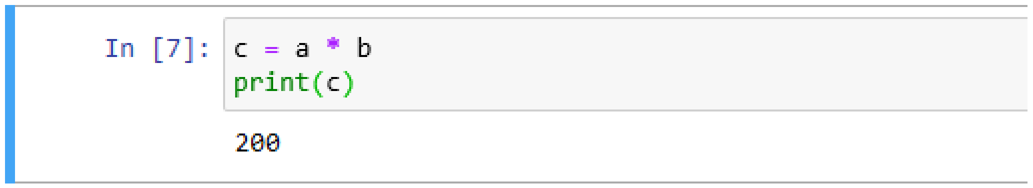
```
In [5]: c = a + b
        print(c)

        30
```

Subtraction:



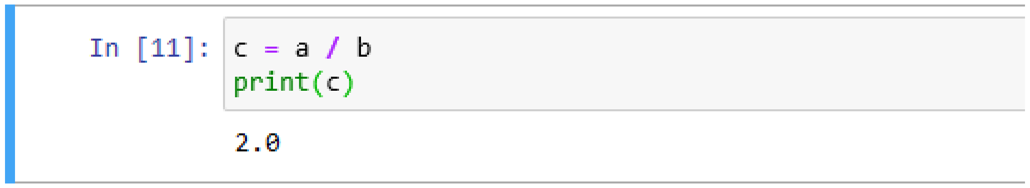
```
In [6]: c = a - b
        print(c)

10
```

Multiplication:



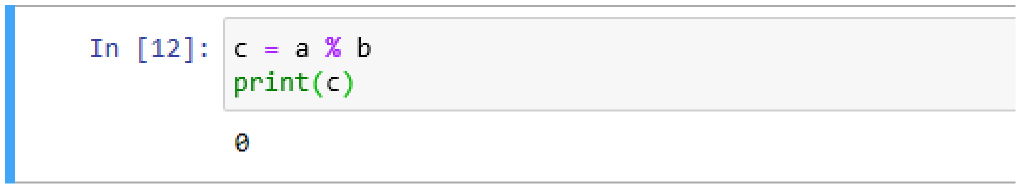
```
In [7]: c = a * b
        print(c)

200
```

Division:



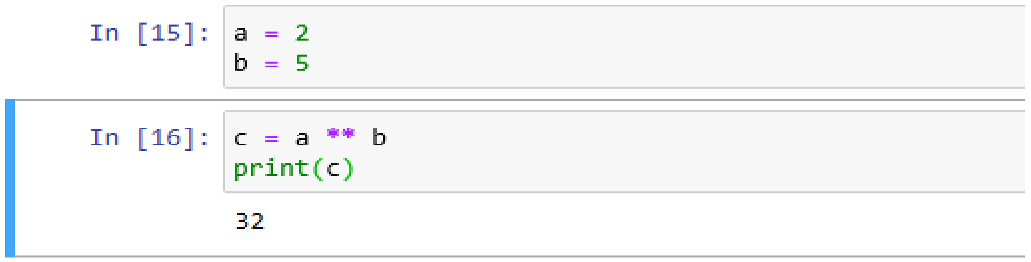
```
In [11]: c = a / b
         print(c)

2.0
```

Remainder:



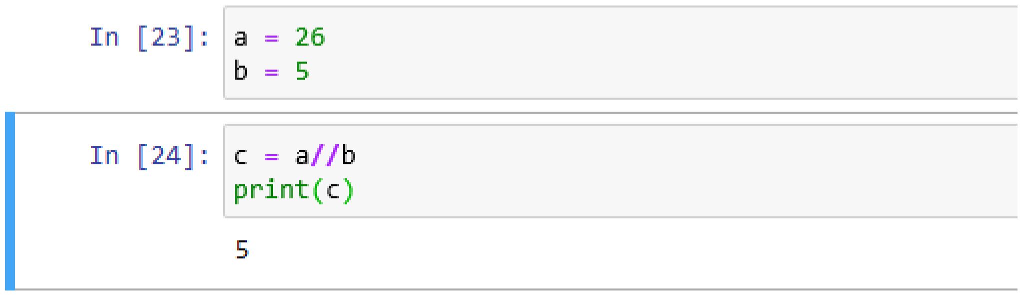
```
In [12]: c = a % b
         print(c)

0
```

Exponent:



```
In [15]: a = 2
         b = 5

In [16]: c = a ** b
         print(c)

32
```

Floor Division:



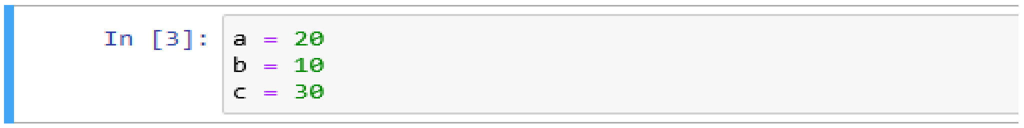
```
In [23]: a = 26
         b = 5

In [24]: c = a//b
         print(c)

5
```

2) Assignment Operators:

The assignment operators are used to **assign** the value of the **right expression** to the **left** operand.

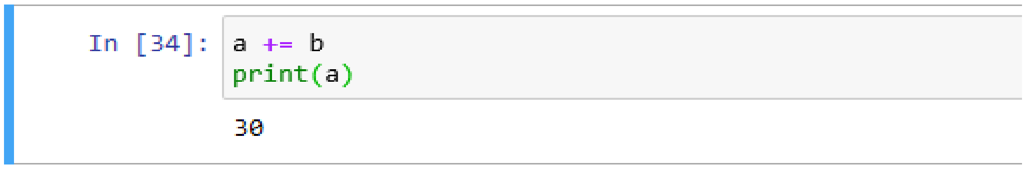
Operator	Description
=	It assigns the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assign the reminder back to left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

E.g. Below variables assigned:



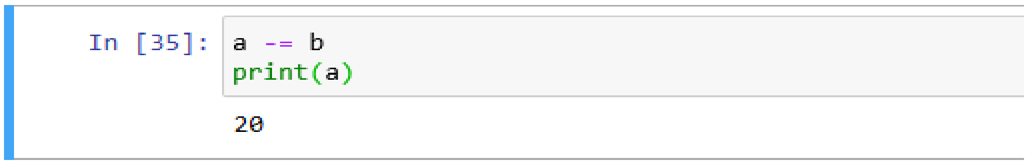
```
In [3]: a = 20
        b = 10
        c = 30
```

Addition Assignment:



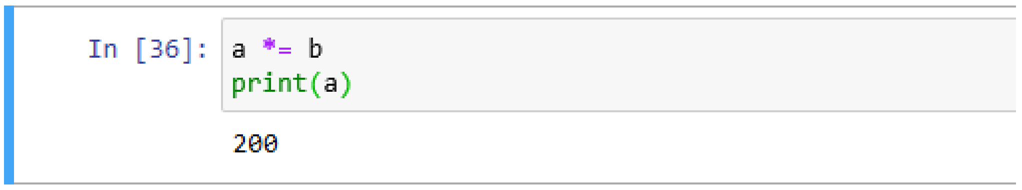
```
In [34]: a += b
         print(a)

         30
```

Subtraction Assignment:



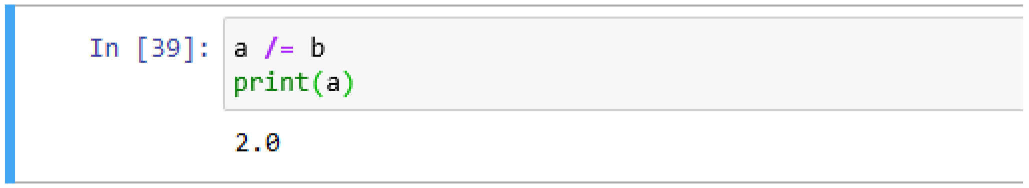
```
In [35]: a -= b
         print(a)

         20
```

Multiplication Assignment:



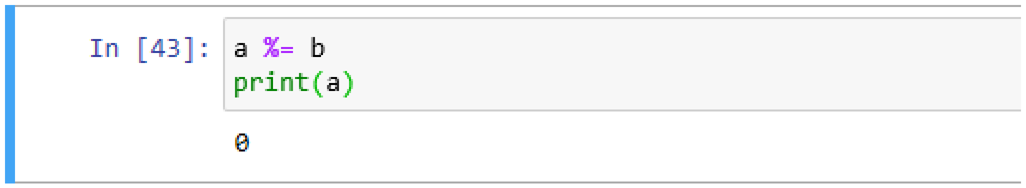
```
In [36]: a *= b
         print(a)

         200
```

Division Assignment:



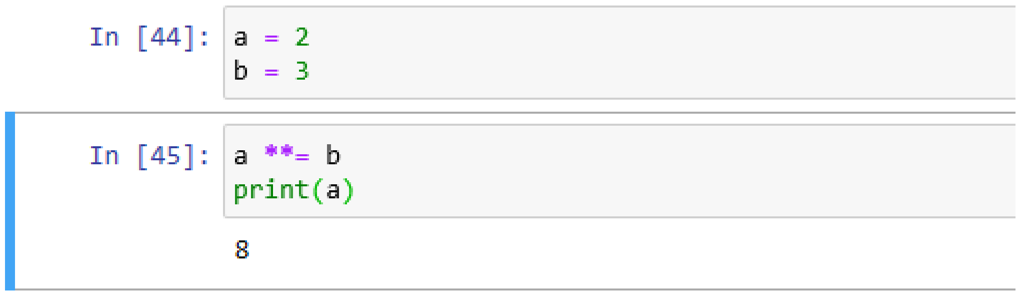
```
In [39]: a /= b
         print(a)

         2.0
```

Remainder Assignment:



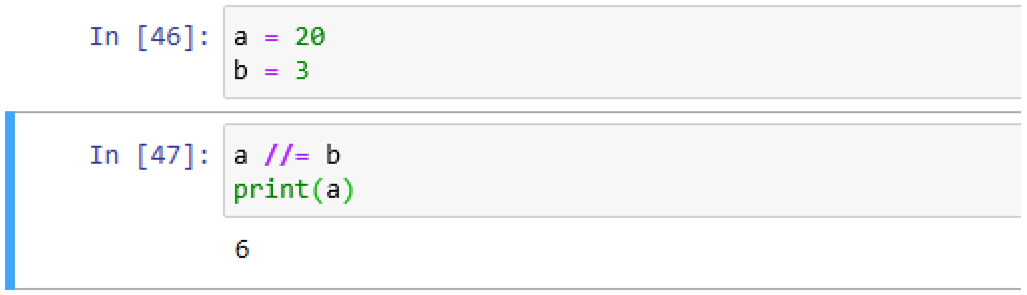
```
In [43]: a %= b
         print(a)

         0
```

Exponent Assignment:



```
In [44]: a = 2
         b = 3

In [45]: a **= b
         print(a)

         8
```

Floor Division Assignment:



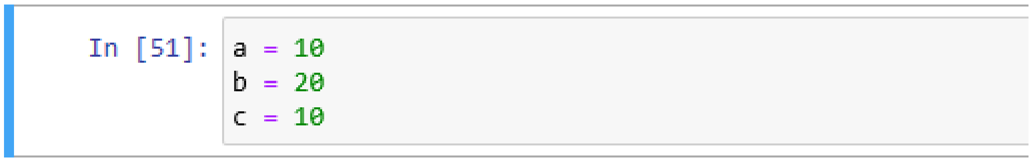
```
In [46]: a = 20
         b = 3

In [47]: a //= b
         print(a)

         6
```

3) Comparison Operators:

Comparison operators are used for **comparing** the **value** of the **two operands** and returns **Boolean** true or false accordingly.

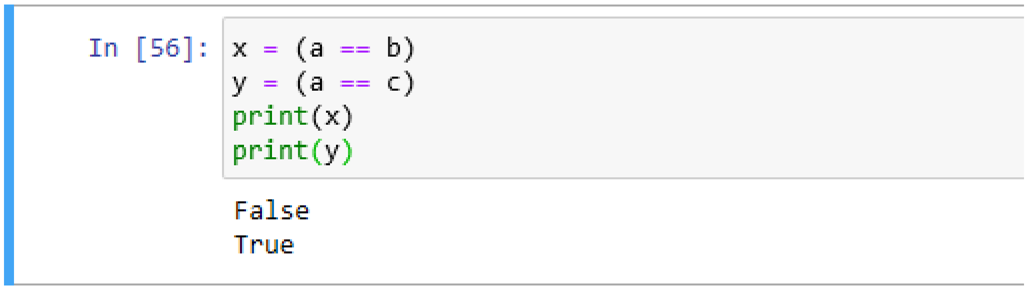
Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

E.g. Below Variables Assigned:



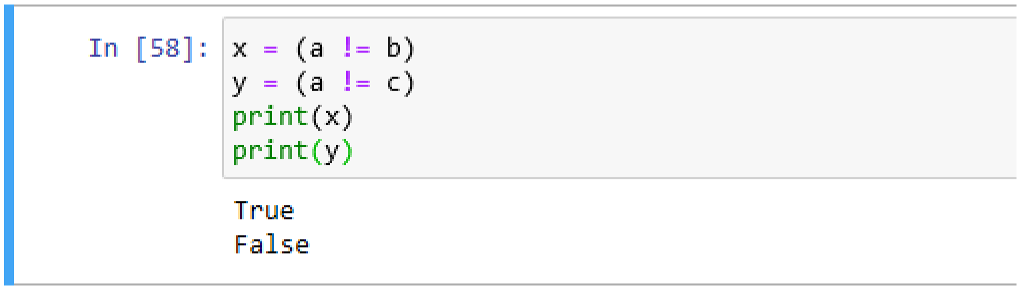
```
In [51]: a = 10
         b = 20
         c = 10
```

Equals:



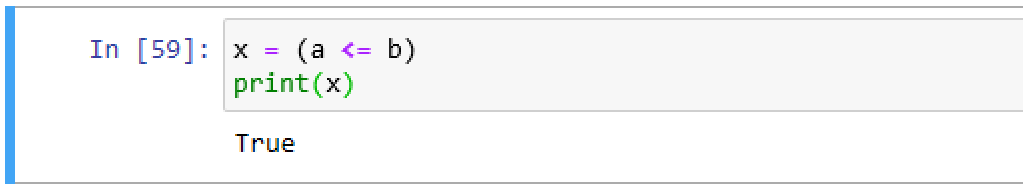
```
In [56]: x = (a == b)
         y = (a == c)
         print(x)
         print(y)

         False
         True
```

Not Equals:

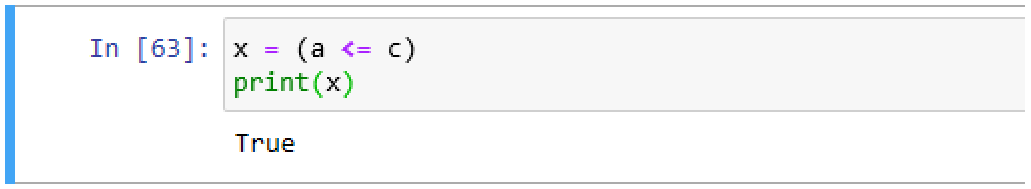


```
In [58]: x = (a != b)
         y = (a != c)
         print(x)
         print(y)

         True
         False
```

Less than Equals:

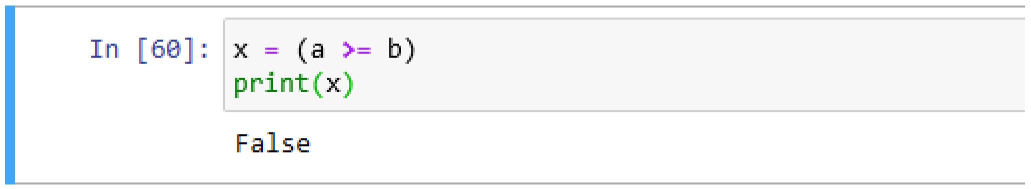


```
In [59]: x = (a <= b)
         print(x)

         True
```

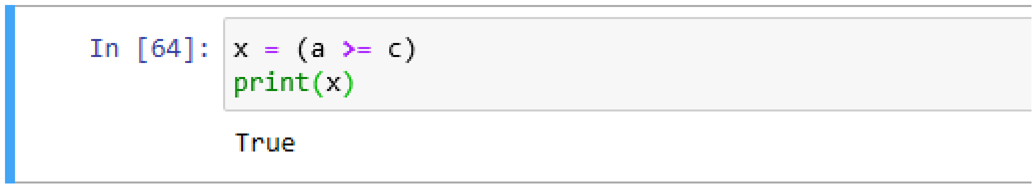


```
In [63]: x = (a <= c)
         print(x)

         True
```

Greater than Equals:

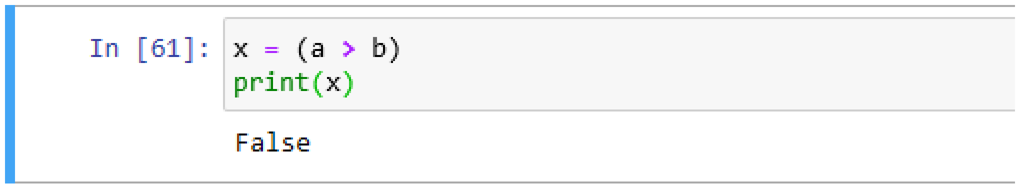


```
In [60]: x = (a >= b)
         print(x)

         False
```



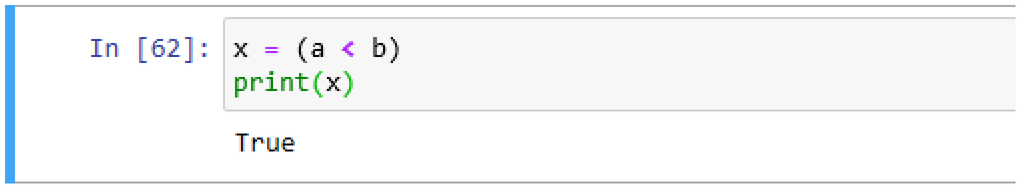
```
In [64]: x = (a >= c)
         print(x)

         True
```

Greater:



```
In [61]: x = (a > b)
         print(x)

         False
```

Lesser:



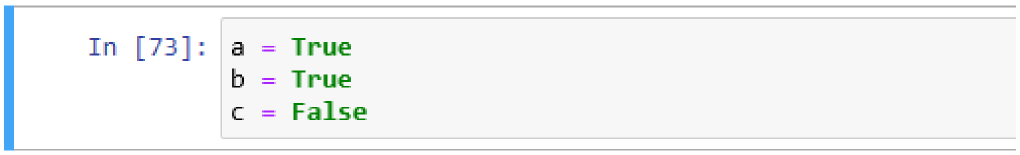
```
In [62]: x = (a < b)
         print(x)

         True
```

4) Logical Operators:

The logical operators are used primarily in **expression evaluation** to make a **decision**.

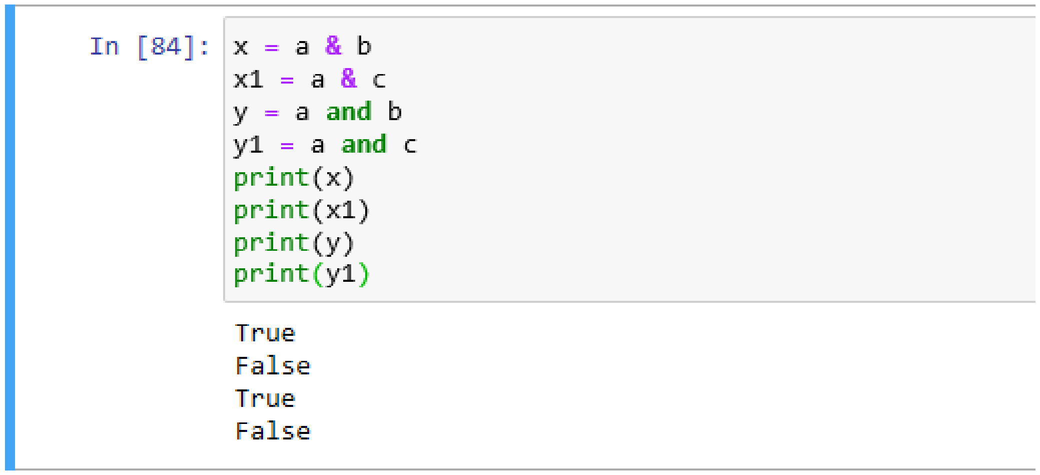
Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression a is true then not (a) will be false and vice versa.

E.g. Below Variables Assigned:



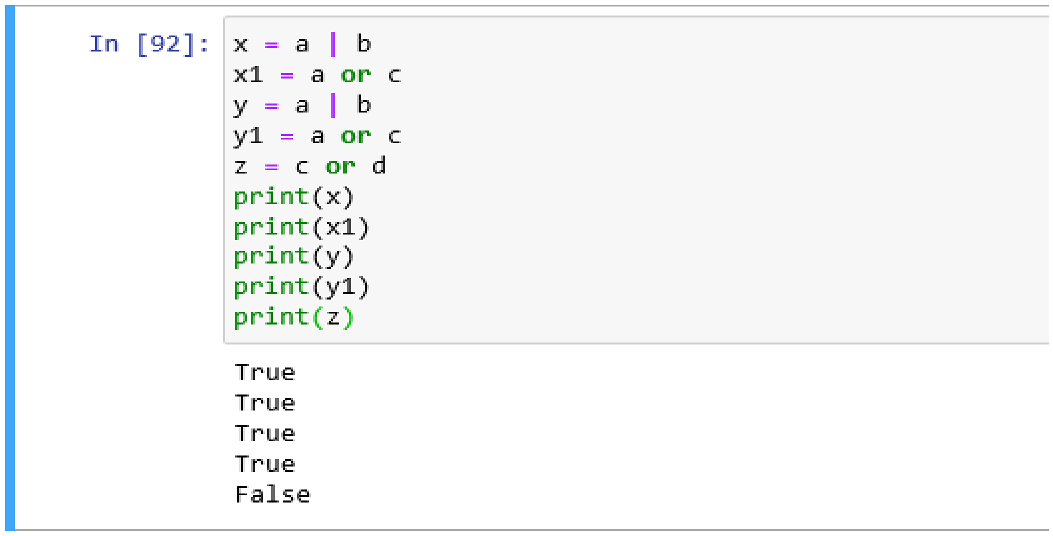
```
In [73]: a = True
         b = True
         c = False
```

AND:



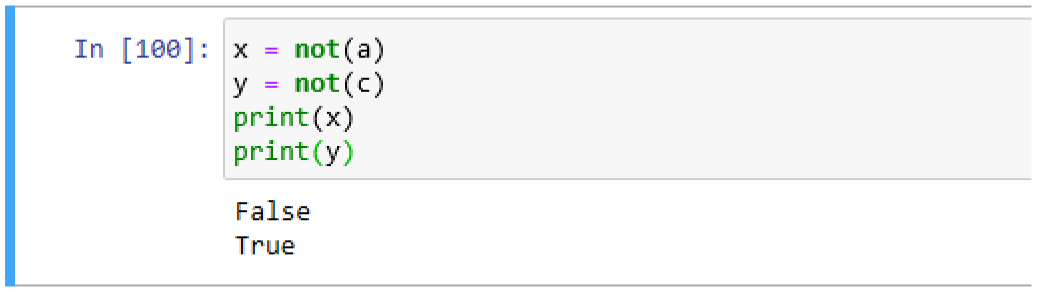
```
In [84]: x = a & b
         x1 = a & c
         y = a and b
         y1 = a and c
         print(x)
         print(x1)
         print(y)
         print(y1)

         True
         False
         True
         False
```

OR:



```
In [92]: x = a | b
         x1 = a or c
         y = a | b
         y1 = a or c
         z = c or d
         print(x)
         print(x1)
         print(y)
         print(y1)
         print(z)

         True
         True
         True
         True
         False
```

NOT:

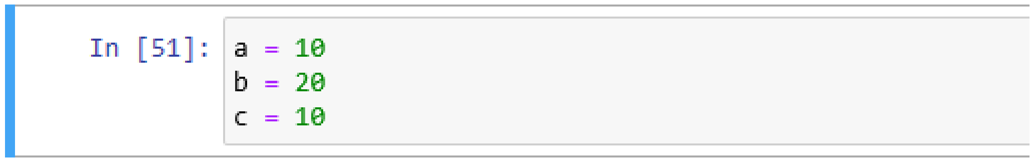


```
In [100]: x = not(a)
          y = not(c)
          print(x)
          print(y)

          False
          True
```

5) Identity Operators:

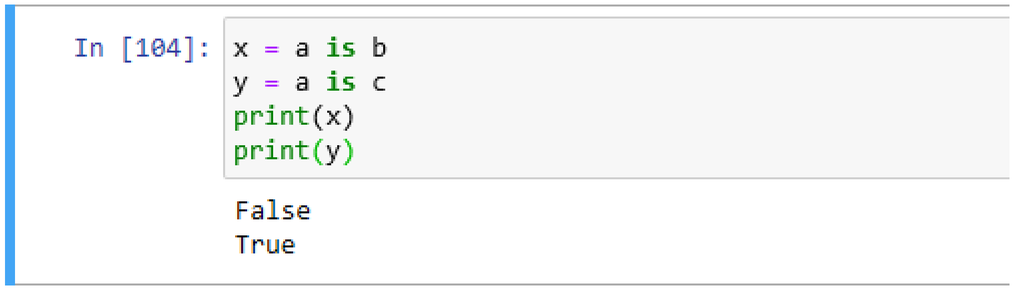
Operator	**Description**
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both side do not point to the same object.

E.g. Below Variables Assigned:



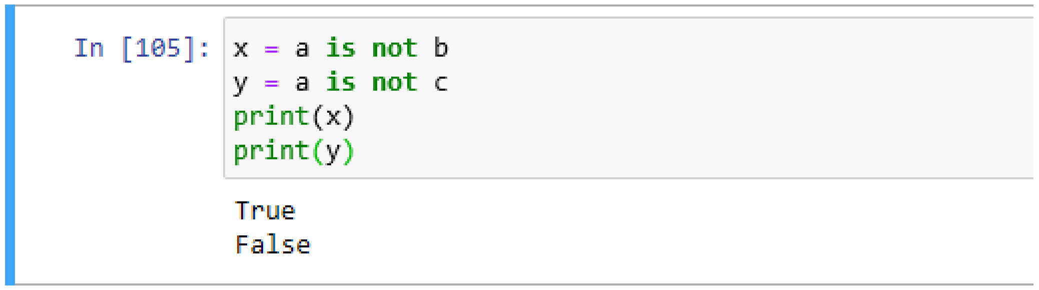
```
In [51]: a = 10
         b = 20
         c = 10
```

IS:



```
In [104]: x = a is b
          y = a is c
          print(x)
          print(y)

          False
          True
```

IS NOT:

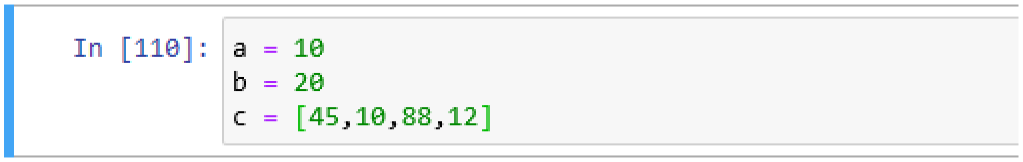


```
In [105]: x = a is not b
          y = a is not c
          print(x)
          print(y)

          True
          False
```

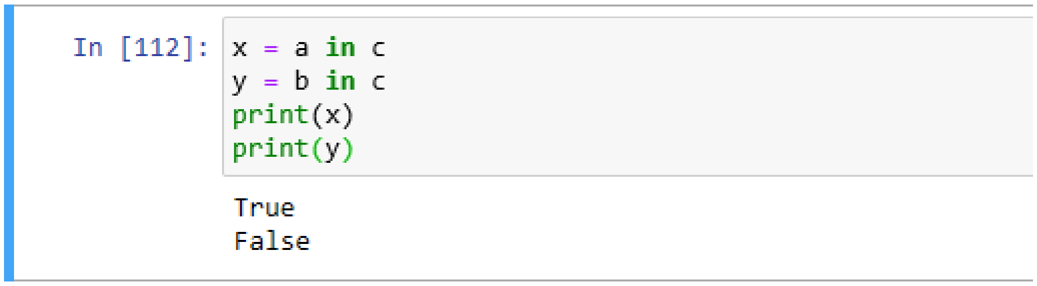
6) Membership Operators:

Operator	**Description**
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

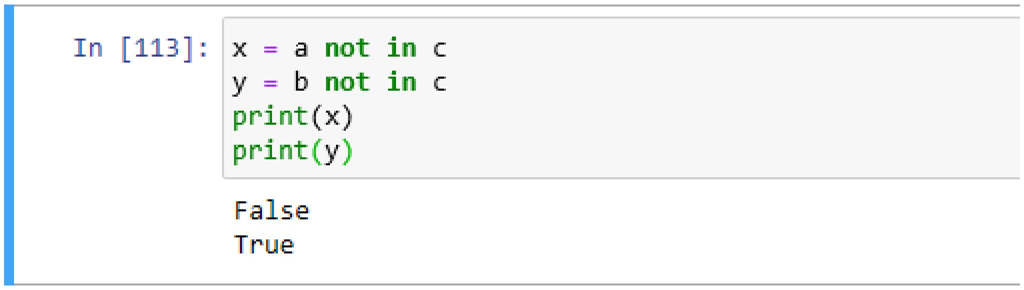
E.g. Below Variables Assigned:


```
In [110]: a = 10
          b = 20
          c = [45,10,88,12]
```

IN:


```
In [112]: x = a in c
          y = b in c
          print(x)
          print(y)

          True
          False
```

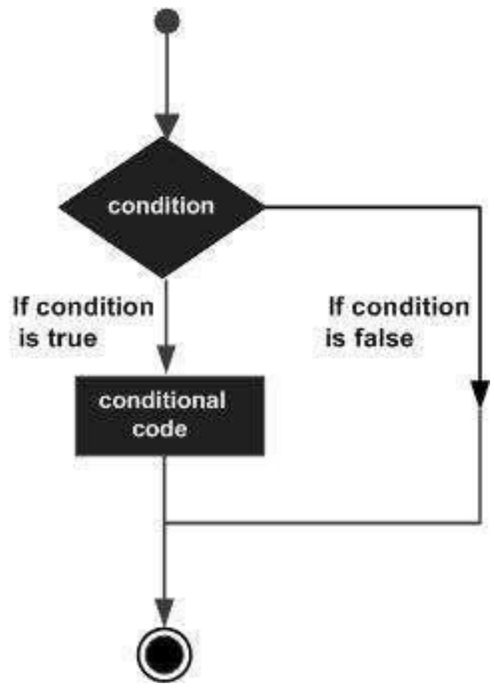
NOT IN:


```
In [113]: x = a not in c
          y = b not in c
          print(x)
          print(y)

          False
          True
```

Decision Making:

- Decision-making is the anticipation of **conditions** occurring during the **execution** of a program and specified **actions** taken according to the **conditions**.
- Decision structures **evaluate** multiple expressions, which produce **TRUE** or **FALSE** as the 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:



Python programming language assumes any **non-zero** and **non-null** values as **TRUE**, and any **zero** or **null** values as **FALSE** value.

Python programming language provides the following types of **decision-making statements**.

Statement	Description
if statements	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
if...else statements	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
nested if statements	Nested if statements enable us to use if else statement inside an outer if statement.

IF Statement:

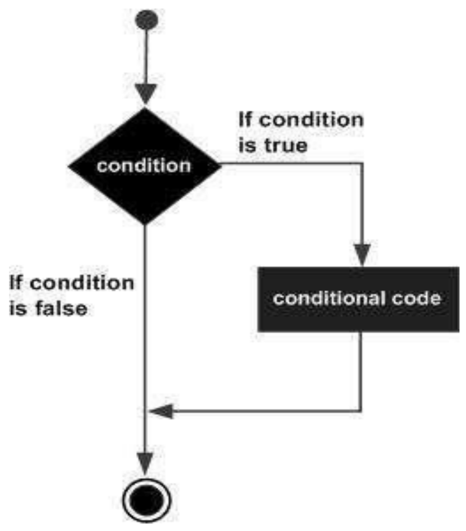
If statement is used to **test** a particular **condition** and if the **condition** is **true**, it **executes** a **block of code** known as **if-block**. The condition of **if statement** can be any **valid** logical expression which can be either **evaluated** to true or false.

Syntax:

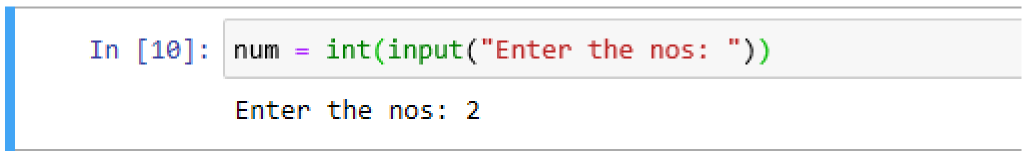
```
if <expression>:
   statement (s)
```

If the Boolean expression **evaluates** to **TRUE**, then the block of statement(s) inside **if statement** is **executed**. In Python, statements in a **block** are **uniformly indented** after the **: symbol**. If Boolean expression evaluates to **FALSE**, then the **first set of code** after the **end of block** is **executed**.

Flow Diagram

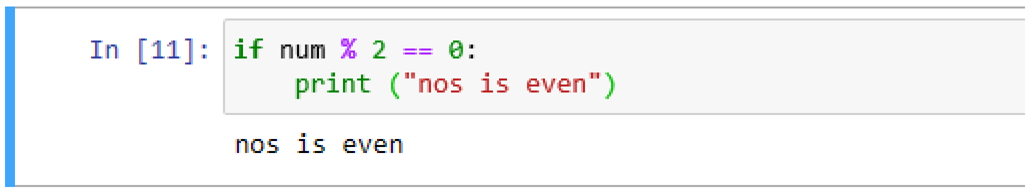


Check the given nos is even:



```
In [10]: num = int(input("Enter the nos: "))

         Enter the nos: 2
```

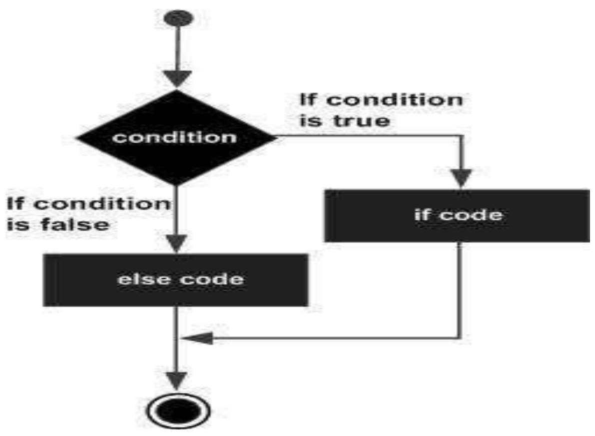
Output:



```
In [11]: if num % 2 == 0:
             print ("nos is even")

         nos is even
```

The if-else statement:

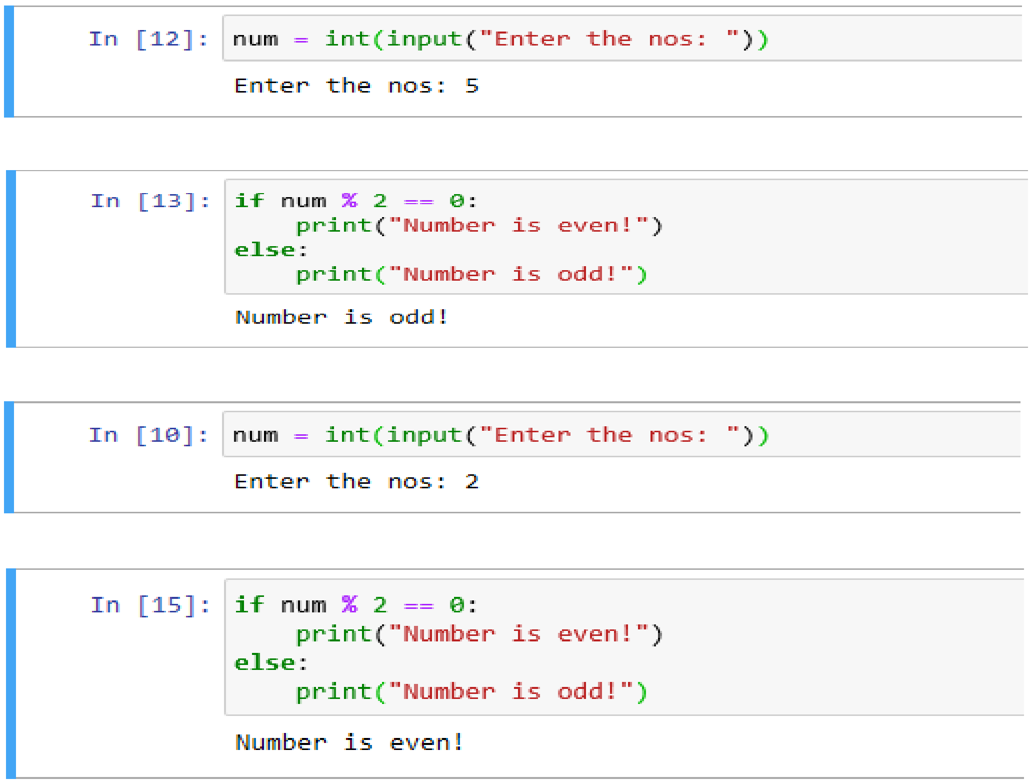
The **if-else** statement provides an **else** block combined with **if statement** which is **executed** in the **false case** of the **condition.** If the condition is **true**, then the **if-block** is **executed**. Otherwise, the **else-block** is **executed.** The **else** statement is an **optional statement** and there could be at the most only **one else statement** following **if.**



Syntax:

if condition:
   #block of statements
else:
   #another block of statements (else-block)

Check the given nos is Even or Odd:



```
In [12]: num = int(input("Enter the nos: "))
         Enter the nos: 5

In [13]: if num % 2 == 0:
             print("Number is even!")
         else:
             print("Number is odd!")
         Number is odd!

In [10]: num = int(input("Enter the nos: "))
         Enter the nos: 2

In [15]: if num % 2 == 0:
             print("Number is even!")
         else:
             print("Number is odd!")
         Number is even!
```

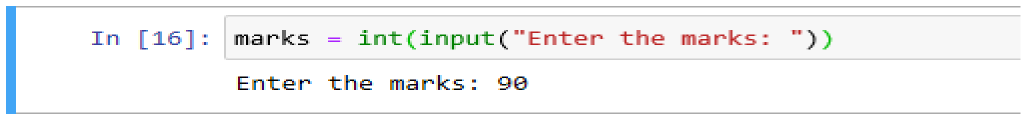
The ELIF Statements:

The **elif** statement enables us to check **multiple conditions** and **execute** the specific **block of statements** depending upon the **true** condition among them. We can have any number of **elif statements** in our program depending upon our need. However, using **elif** is **optional**.

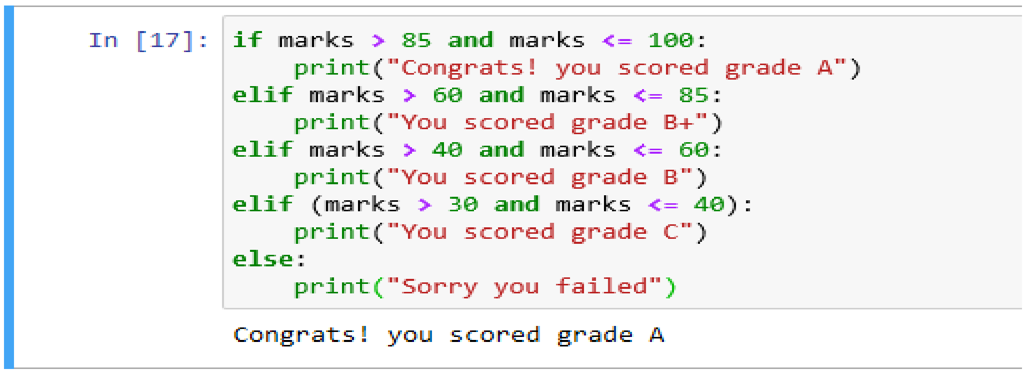
Syntax:

```
if expression 1:
   # block of statements
elif expression 2:
   # block of statements
elif expression 3:
   # block of statements
else:
   # block of statements
```

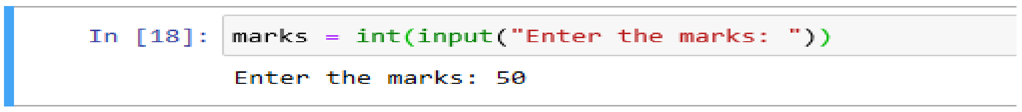
Checking the grade of a student:



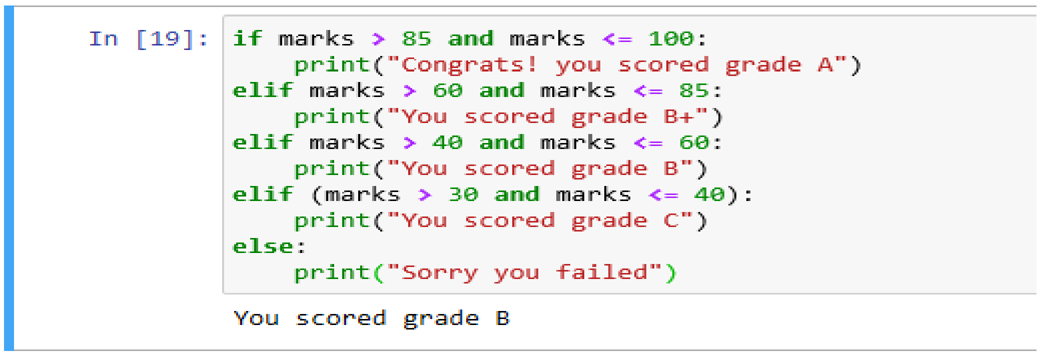
```
In [16]: marks = int(input("Enter the marks: "))

         Enter the marks: 90
```



```
In [17]: if marks > 85 and marks <= 100:
             print("Congrats! you scored grade A")
         elif marks > 60 and marks <= 85:
             print("You scored grade B+")
         elif marks > 40 and marks <= 60:
             print("You scored grade B")
         elif (marks > 30 and marks <= 40):
             print("You scored grade C")
         else:
             print("Sorry you failed")

         Congrats! you scored grade A
```



```
In [18]: marks = int(input("Enter the marks: "))

         Enter the marks: 50
```



```
In [19]: if marks > 85 and marks <= 100:
             print("Congrats! you scored grade A")
         elif marks > 60 and marks <= 85:
             print("You scored grade B+")
         elif marks > 40 and marks <= 60:
             print("You scored grade B")
         elif (marks > 30 and marks <= 40):
             print("You scored grade C")
         else:
             print("Sorry you failed")

         You scored grade B
```

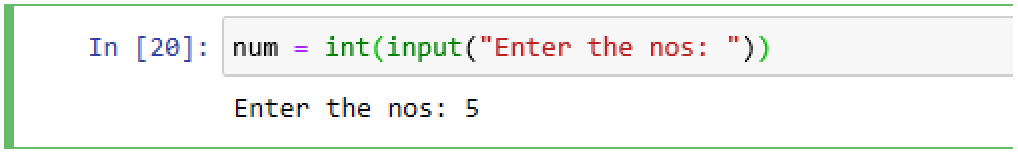
Nested IF Statements:

There may be a situation when you want to check for **another condition after** a **condition resolves** to **true**. In such a situation, you can use the **nested if construct**.
In a **nested if construct**, you can have **if...elif...else** construct inside another **if...elif...else** construct.

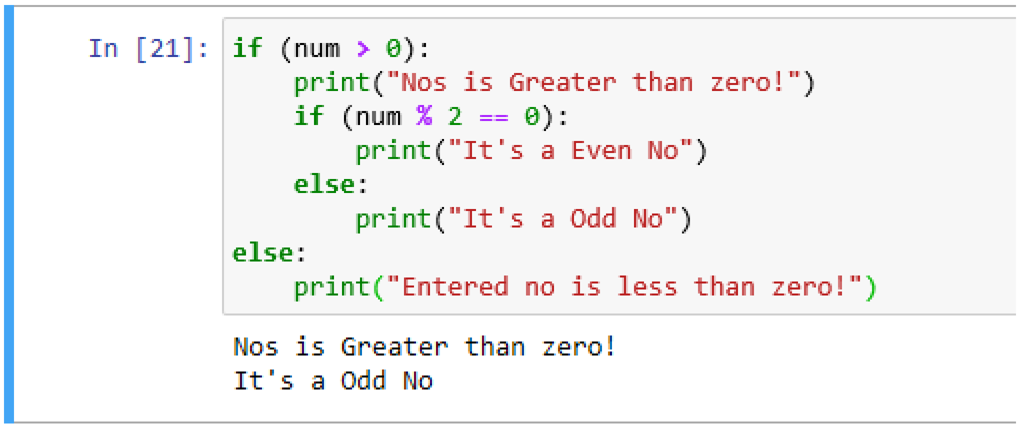
Syntax:

```
If expression1:
   #statement(s)
   if expression2:
      #statement(s)
      elif expression3:
           #statement(s)
   else:
   #statement(s)
else:
   #statement(s)
```

Check the given nos is Even or Odd:



```
In [20]: num = int(input("Enter the nos: "))

         Enter the nos: 5
```

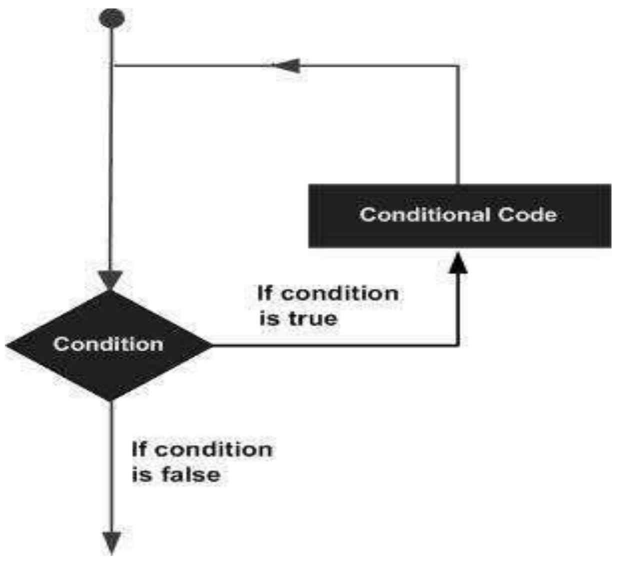


```
In [21]: if (num > 0):
             print("Nos is Greater than zero!")
             if (num % 2 == 0):
                 print("It's a Even No")
             else:
                 print("It's a Odd No")
         else:
             print("Entered no is less than zero!")

         Nos is Greater than zero!
         It's a Odd No
```

Python Loops:

In general, **statements** are **executed sequentially**. The **first statement** in a **function** is **executed** first, followed by the **second**, and so on. There may be a situation when you need to **execute** a **block of code several number of times**. Hence, **loop statement allows** us to execute a **statement** or **group of statements multiple times**.

The following diagram illustrates a **loop statement**:



Advantages of loops:

There are the following advantages of loops in Python.

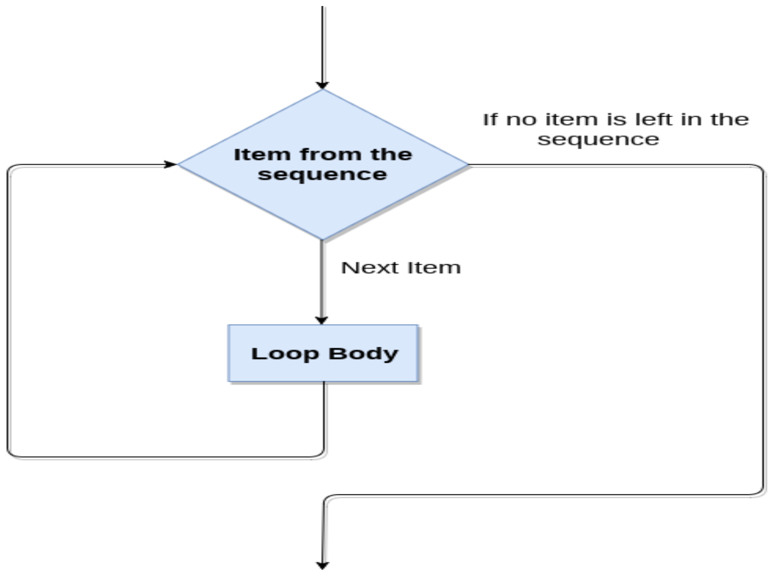
- It provides code re-usability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of data structures.

Loop Statements in Python:

Loop Type	Description
for loop	for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.
while loop	The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.
do-while loop	The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once.

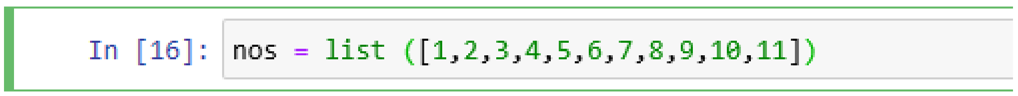
for loop:
for **loop in Python** is used to **iterate** the statements or a **part** of the program **several times**. It is **frequently** used to **traverse** the **data structures** like **list**, **tuple**, **string** or **dictionary**.

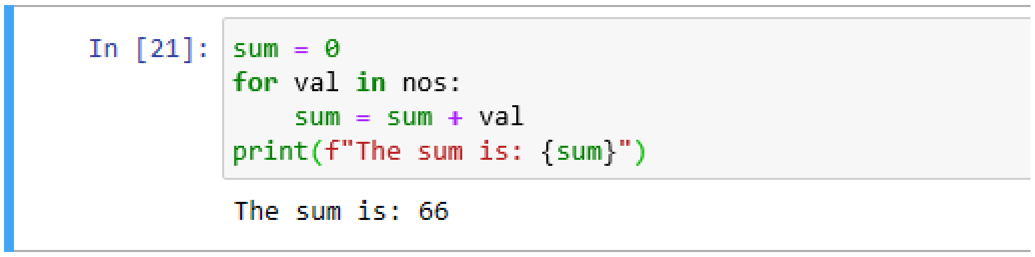
Syntax:
```
for iterating_var in sequence:
      statement (s)
```



E.g. Program to find the sum of all numbers stored in a list

```
nos = [1,2,3,4,5,6,7,8,9,10,11]
sum = 0
for val in nos:
          sum = sum + val
print ("The sum is", sum)
```

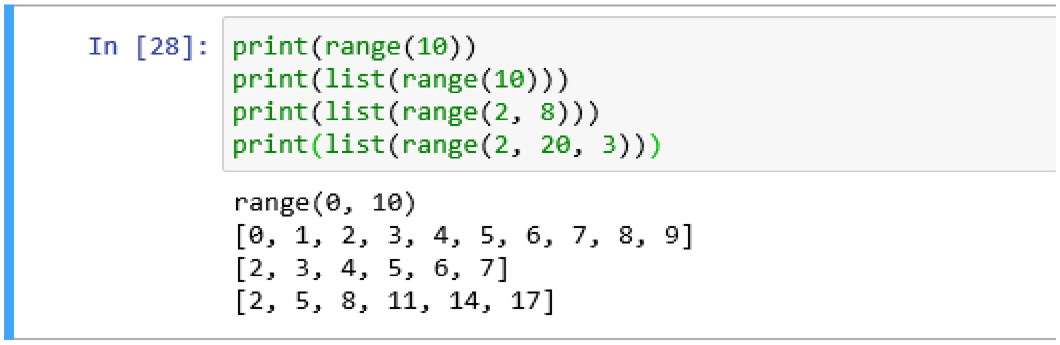

```
In [16]: nos = list ([1,2,3,4,5,6,7,8,9,10,11])
```


```
In [21]: sum = 0
         for val in nos:
             sum = sum + val
         print(f"The sum is: {sum}")

         The sum is: 66
```

The range ( ) function:

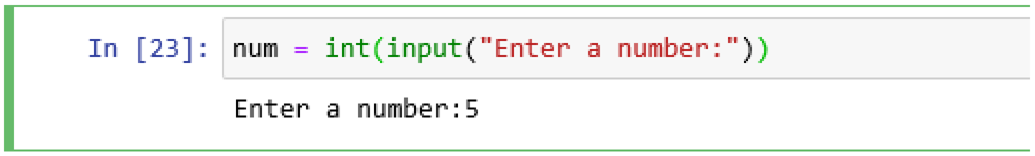
We can generate a **sequence** of numbers using **range ( )** function. range (10) will generate numbers from 0 to 9 (10 numbers). We can also define the **start**, **stop** and **step** size as **range (start, stop, step_size)**. Default **step_size** is **1** if **not** provided.

The following example will clarify this:

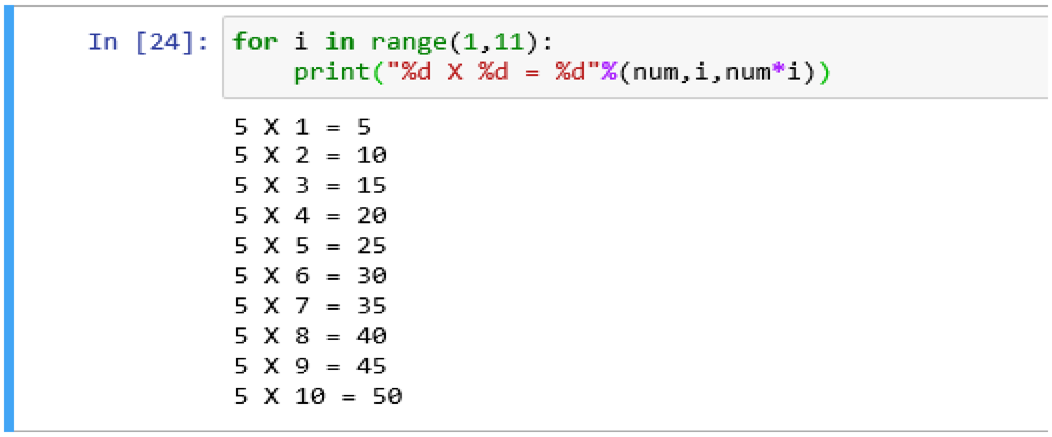


```
In [28]: print(range(10))
         print(list(range(10)))
         print(list(range(2, 8)))
         print(list(range(2, 20, 3)))

         range(0, 10)
         [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
         [2, 3, 4, 5, 6, 7]
         [2, 5, 8, 11, 14, 17]
```

Printing a table:

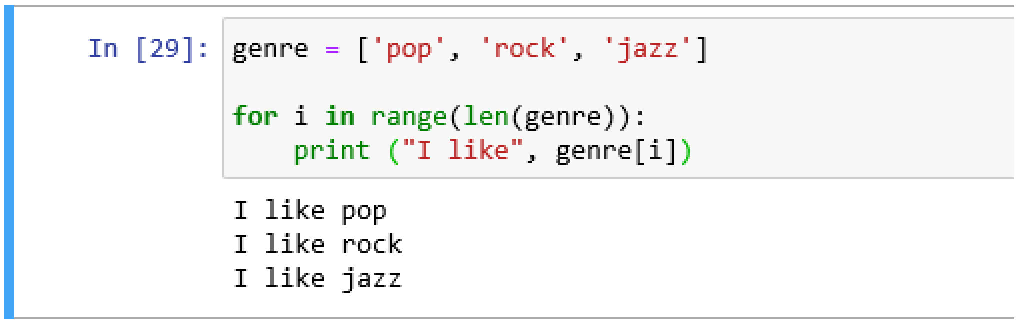


```
In [23]: num = int(input("Enter a number:"))

         Enter a number:5
```



```
In [24]: for i in range(1,11):
             print("%d X %d = %d"%(num,i,num*i))

         5 X 1 = 5
         5 X 2 = 10
         5 X 3 = 15
         5 X 4 = 20
         5 X 5 = 25
         5 X 6 = 30
         5 X 7 = 35
         5 X 8 = 40
         5 X 9 = 45
         5 X 10 = 50
```

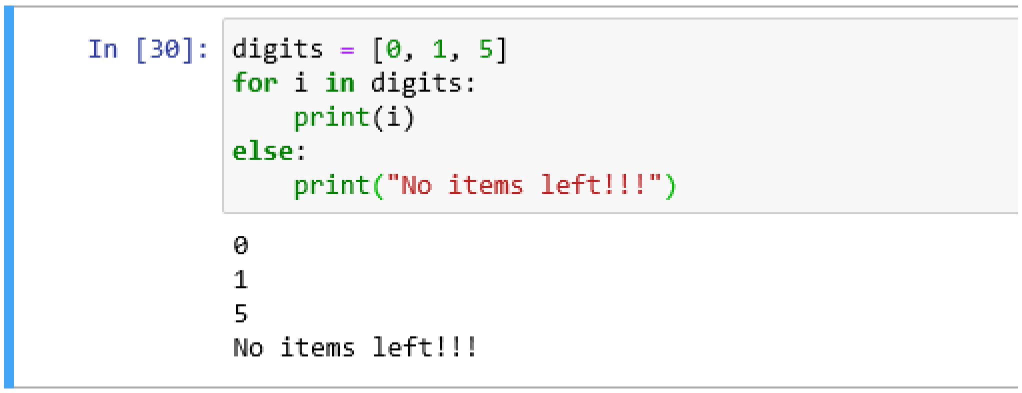
We can use **range ( )** function in **for** loops to **iterate** through a **sequence** of numbers. It can be combined with the **len ( )** function to **iterate** through a **sequence** using **indexing**.



```
In [29]: genre = ['pop', 'rock', 'jazz']

         for i in range(len(genre)):
             print ("I like", genre[i])

         I like pop
         I like rock
         I like jazz
```

for loop with else:

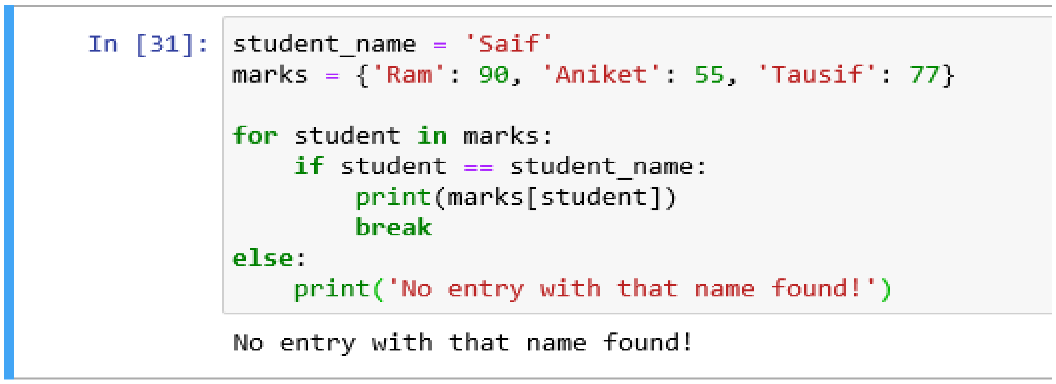
A **for** loop can have an **optional else** block as well. The **else** part is **executed** if the items in the **sequence** used in **for** loop **exhausts**. The **break** keyword can be used to **stop** for loop. In such cases, the **else** part is ignored. Hence, for loop's else part runs if **no**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



```
In [30]: digits = [0, 1, 5]
         for i in digits:
             print(i)
         else:
             print("No items left!!!")

         0
         1
         5
         No items left!!!
```

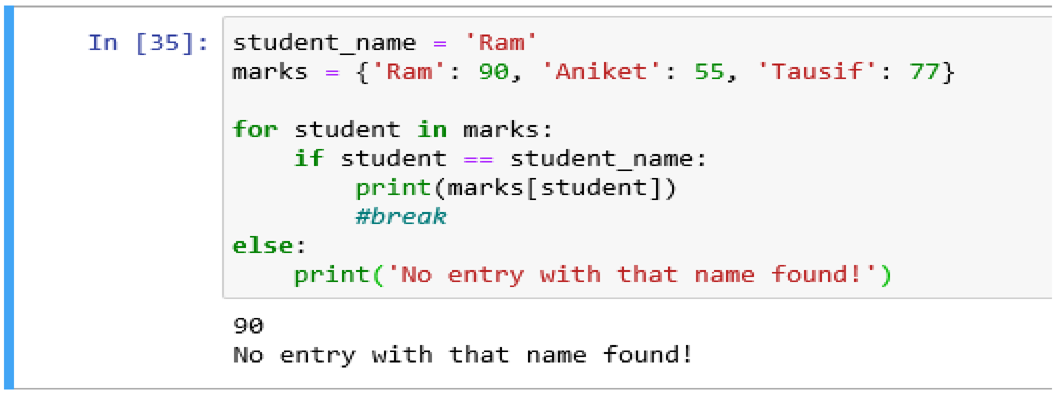
Here, **for loop** prints items of the **list** until the loop **exhausts**. When the for loop exhausts, it **executes** the block of code in the **else** and prints No items left. This **for...else** statement can be used with the **break keyword** to **run** the **else block** only when the **break** keyword was **not** executed.

With Break Statement:



```
In [31]: student_name = 'Saif'
         marks = {'Ram': 90, 'Aniket': 55, 'Tausif': 77}

         for student in marks:
             if student == student_name:
                 print(marks[student])
                 break
         else:
             print('No entry with that name found!')

         No entry with that name found!
```

Without Break Statement:

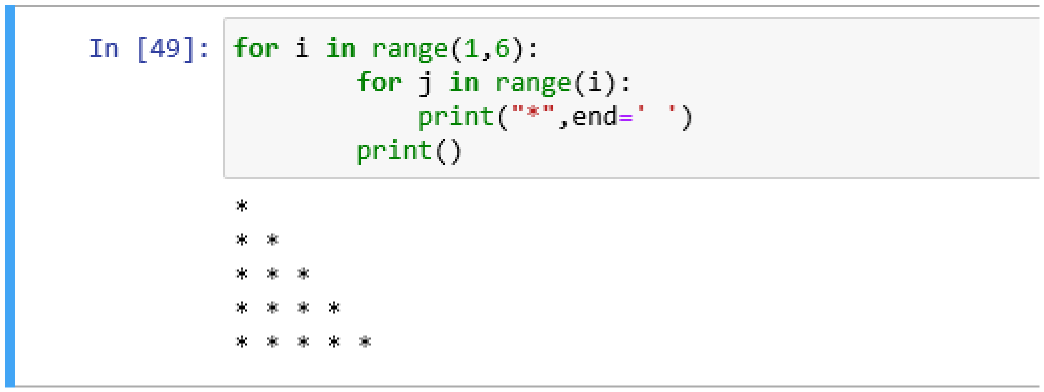


```
In [35]: student_name = 'Ram'
         marks = {'Ram': 90, 'Aniket': 55, 'Tausif': 77}

         for student in marks:
             if student == student_name:
                 print(marks[student])
                 #break
         else:
             print('No entry with that name found!')

         90
         No entry with that name found!
```

Nested for loop:
Python allows us to **nest any number** of **for** loops inside **a for** loop. The **inner** loop is **executed n number of times** for every **iteration** of the outer loop. The syntax of the nested for loop in python is given below.

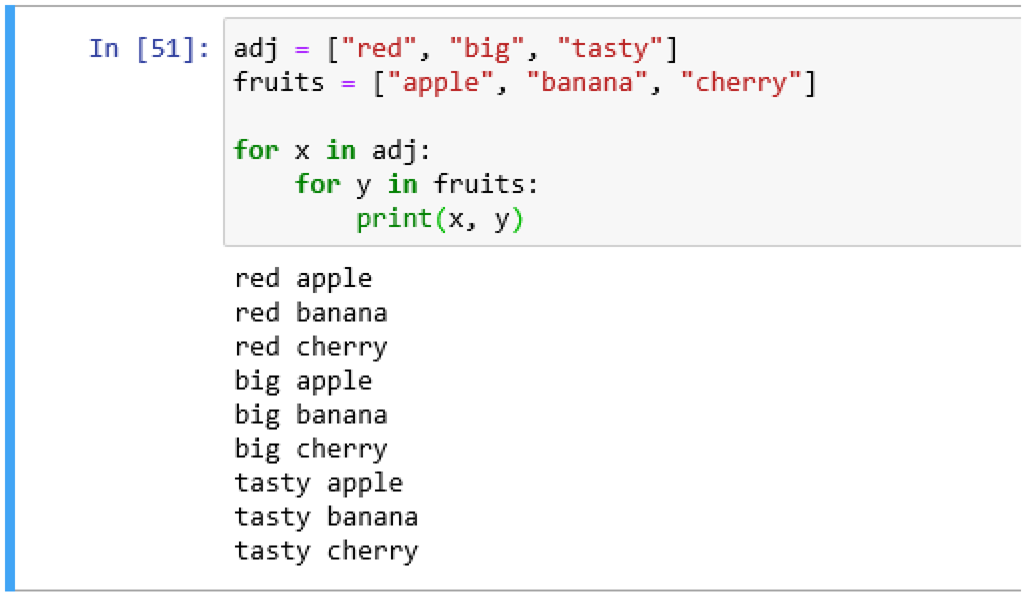
Syntax:

```
for iterating_var1 in sequence:
   for iterating_var2 in sequence:
      #block of statements
#other statements
```



```
In [49]: for i in range(1,6):
                 for j in range(i):
                     print("*",end=' ')
                 print()
```

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



```
In [51]: adj = ["red", "big", "tasty"]
         fruits = ["apple", "banana", "cherry"]

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

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

while loop:

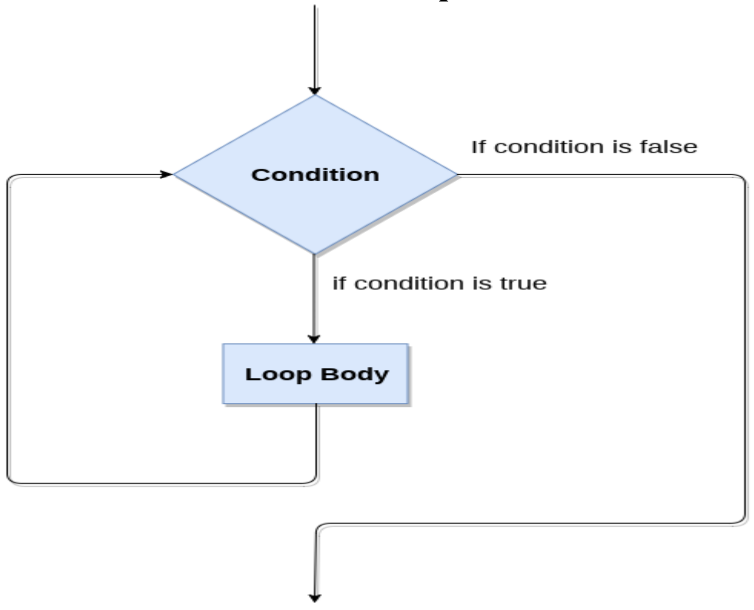
The **while loop** is also known as a **pre-tested** loop. In general, a while loop **allows** a part of the code to be executed as long as the given **condition is true**. It can be viewed as a **repeating if statement**. The while loop is mostly used in the case where the **number** of **iterations** is **not known** in advance.

Syntax:

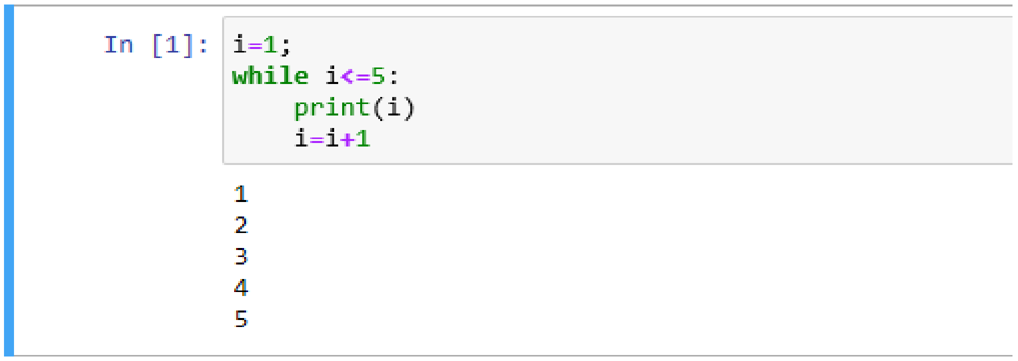
```
while test_expression:
    statement (s)
```

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

In Python, the **body** of the **while** loop is determined through **indentation**. The body starts with **indentation** and the **first unindented** line marks the end. Python **interprets** any **non-zero** value as **True**. **None** and **0** are **interpreted** as **False**.



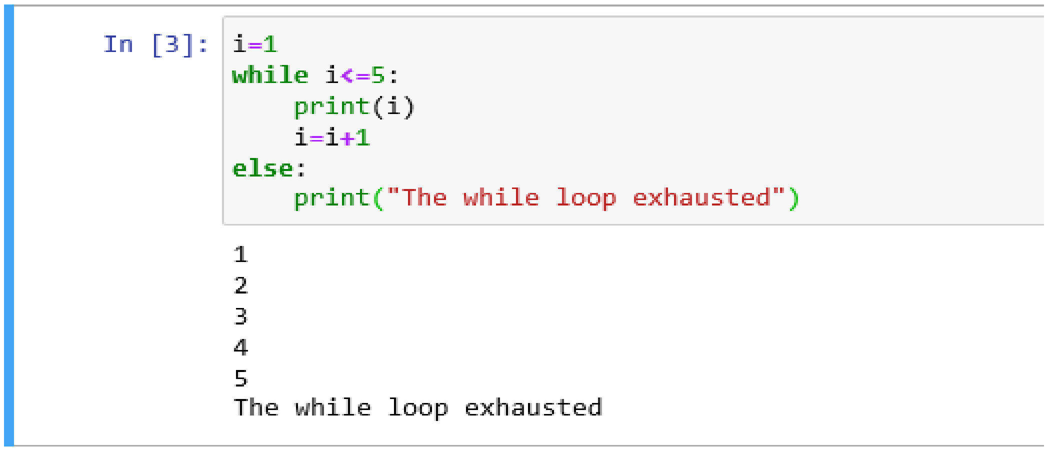
Print natural nos 1 to 5:



```
In [1]: i=1;
        while i<=5:
            print(i)
            i=i+1

1
2
3
4
5
```

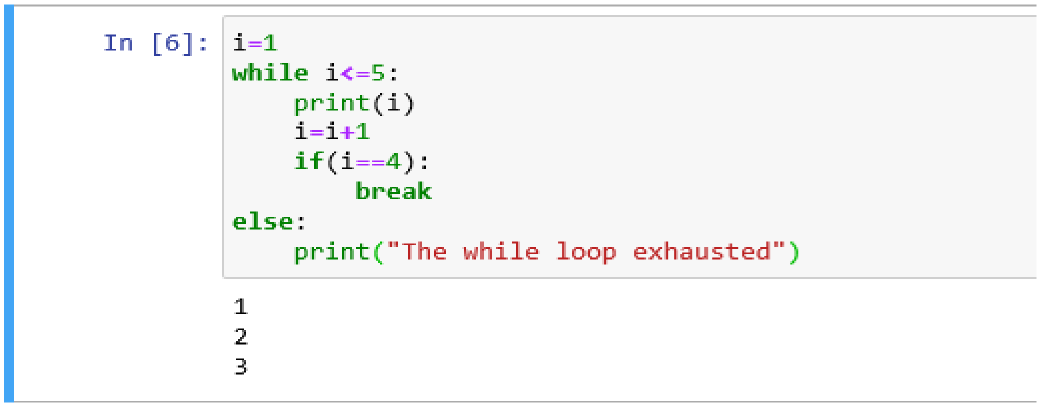
Using else with while loop:
Python **enables** us to use the **while loop** with the **else loop** also. The **else block** is **executed** when the **condition** given in the **while statement** becomes **false**. Like for loop, if the while loop is **broken** using **break** statement, then the **else** block will not be **executed** and the statement present after **else** block will be executed.

Without break:



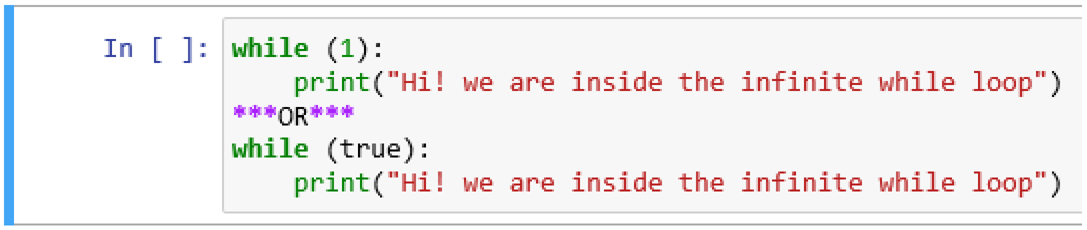
```
In [3]: i=1
        while i<=5:
            print(i)
            i=i+1
        else:
            print("The while loop exhausted")

        1
        2
        3
        4
        5
        The while loop exhausted
```

With break:



```
In [6]: i=1
        while i<=5:
            print(i)
            i=i+1
            if(i==4):
                break
        else:
            print("The while loop exhausted")

        1
        2
        3
```

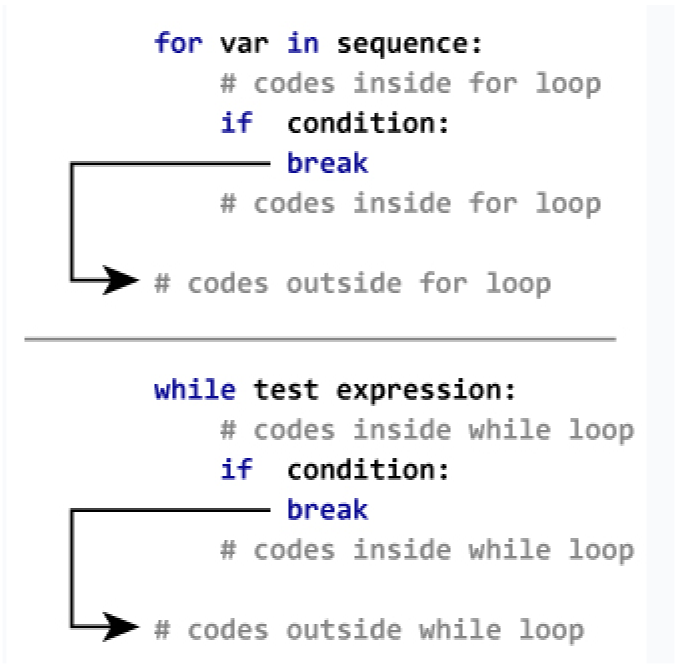
Infinite while loop:
If the condition given in the while loop **never** becomes **false** then the while loop will **never terminate** and result into the **infinite** while loop. Any **non-zero** value in the while loop indicates an **always-true** condition whereas **0** indicates the **always-false** condition. This type of approach is useful if we want our program to **run continuously** in the loop without any disturbance.



```
In [ ]: while (1):
            print("Hi! we are inside the infinite while loop")
        ***OR***
        while (true):
            print("Hi! we are inside the infinite while loop")
```

break statement:

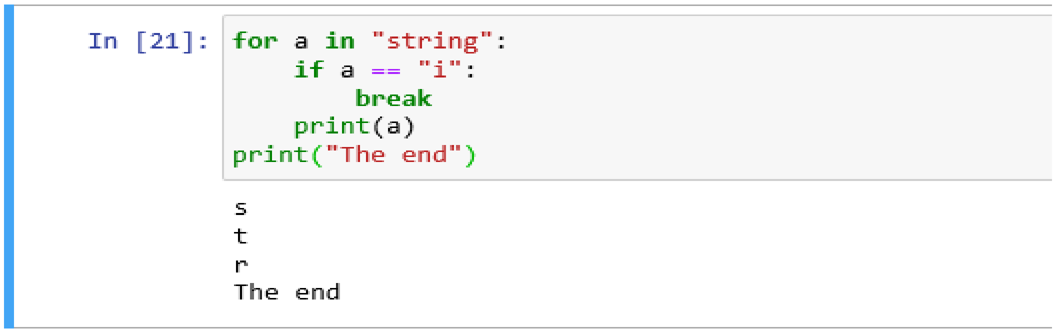
The **break** is a keyword in python which is used to bring the **program control out** of the loop. The break statement **breaks** the loops **one by one**, i.e., in the case of **nested** loops, it breaks the **inner loop first** and then proceeds to **outer** loops. In other words, we can say that break is used to **abort** the **current execution** of the program and the **control** goes to the **next line after** the loop.



```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

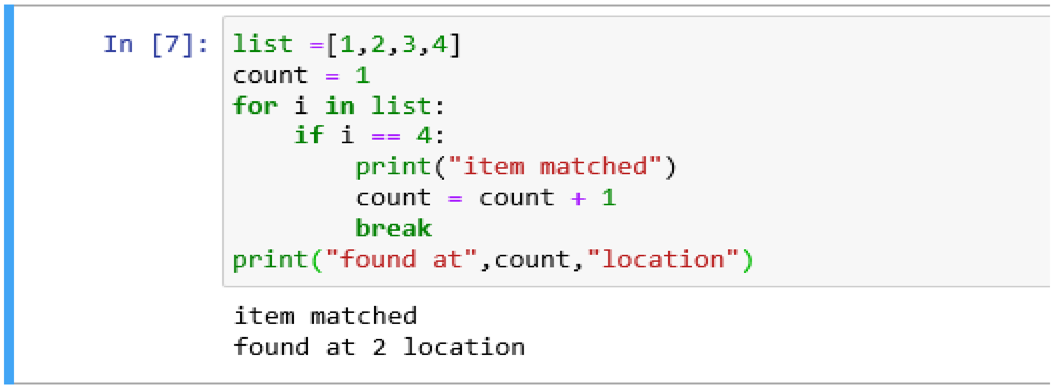
```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

break statement with for loop:



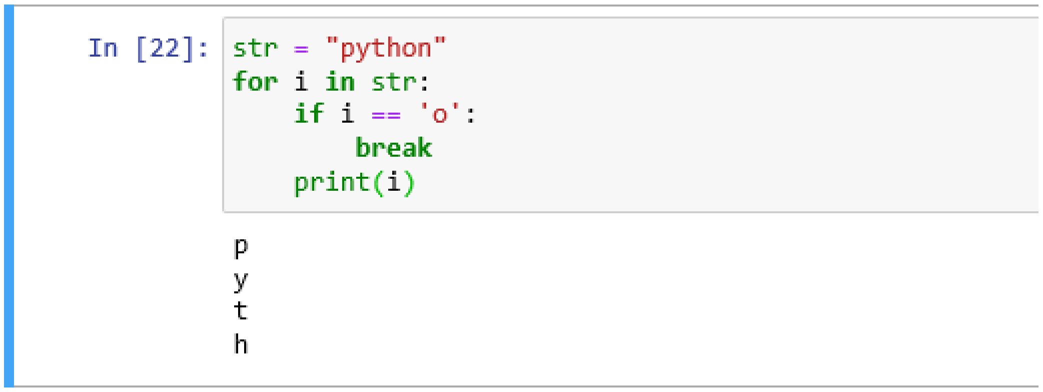
```
In [21]: for a in "string":
             if a == "i":
                 break
             print(a)
         print("The end")
```

```
s
t
r
The end
```



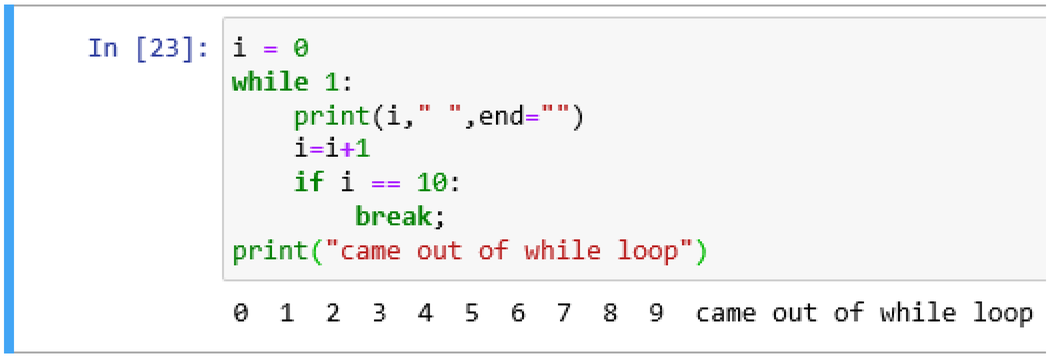
```
In [7]: list =[1,2,3,4]
        count = 1
        for i in list:
            if i == 4:
                print("item matched")
                count = count + 1
                break
        print("found at",count,"location")
```

```
item matched
found at 2 location
```

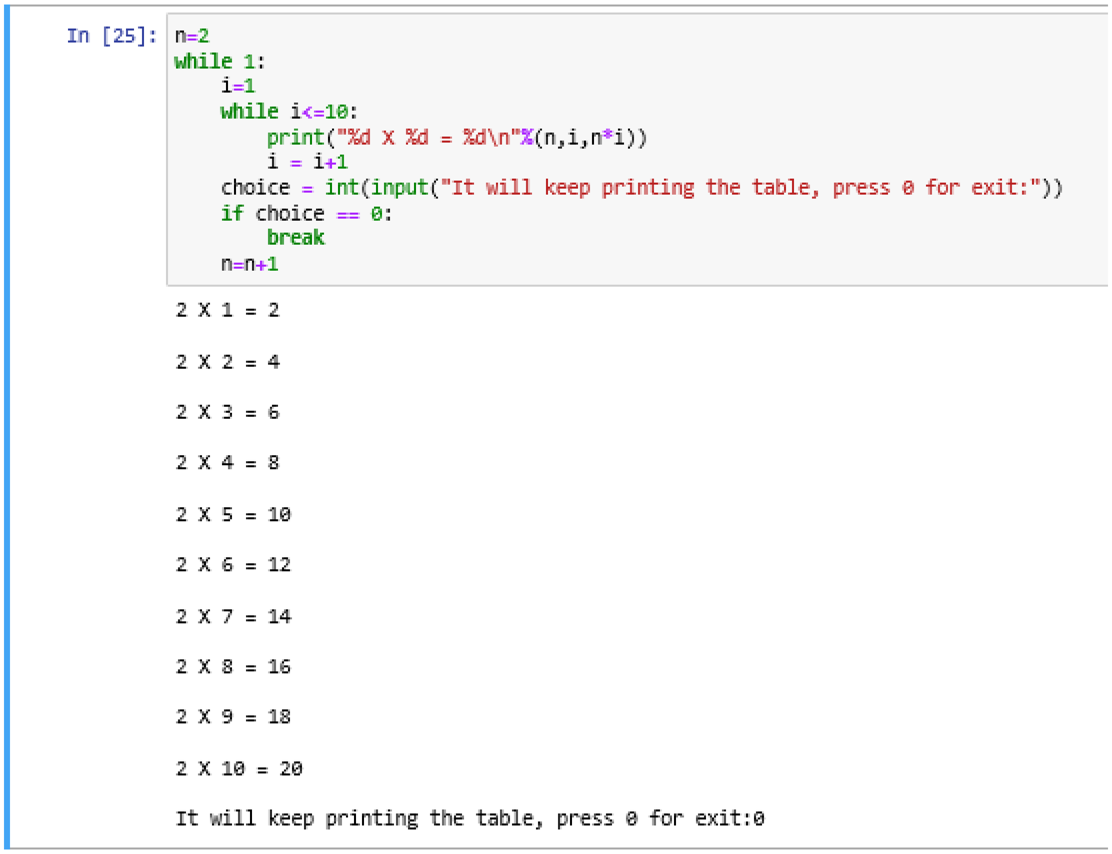


```
In [22]: str = "python"
         for i in str:
             if i == 'o':
                 break
             print(i)

p
y
t
h
```

break statement with for loop:

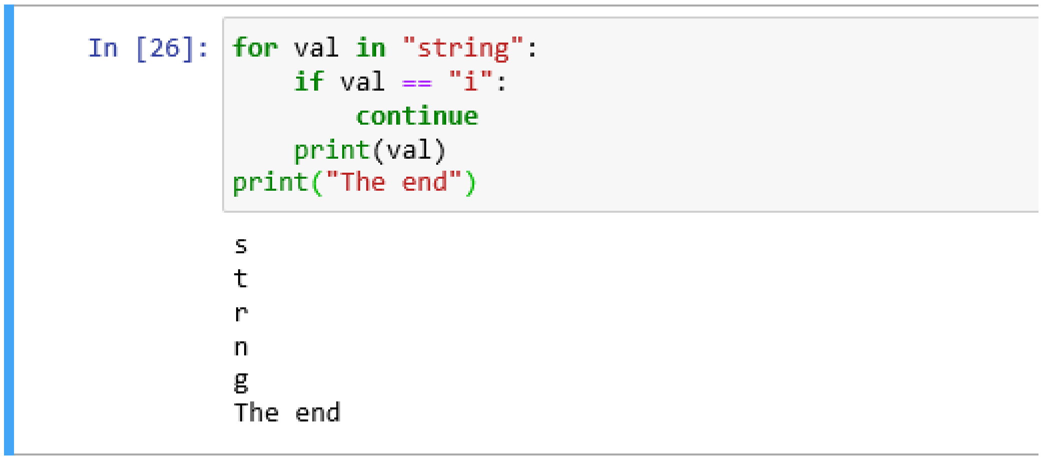


```
In [23]: i = 0
         while 1:
             print(i," ",end="")
             i=i+1
             if i == 10:
                 break;
         print("came out of while loop")

0  1  2  3  4  5  6  7  8  9  came out of while loop
```



```
In [25]: n=2
         while 1:
             i=1
             while i<=10:
                 print("%d X %d = %d\n"%(n,i,n*i))
                 i = i+1
             choice = int(input("It will keep printing the table, press 0 for exit:"))
             if choice == 0:
                 break
             n=n+1

2 X 1 = 2

2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

It will keep printing the table, press 0 for exit:0
```

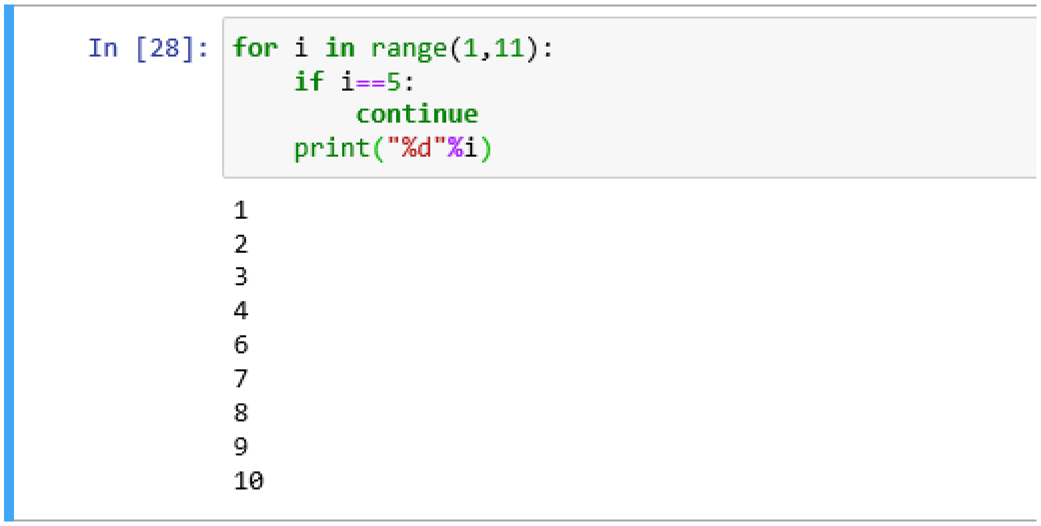
continue Statement:
The **continue** statement is used to **skip** the rest of the code **inside** a loop for the **current iteration** only. Loop does not **terminate** but **continues on** with the **next iteration**.



```
In [26]: for val in "string":
             if val == "i":
                 continue
             print(val)
         print("The end")
```

```
s
t
r
n
g
The end
```

We **continue** with the loop, if the string is **i**, not executing the rest of the block. Hence, we see in our output that all the letters **except i** gets printed.



```
In [28]: for i in range(1,11):
             if i==5:
                 continue
             print("%d"%i)
```

```
1
2
3
4
6
7
8
9
10
```

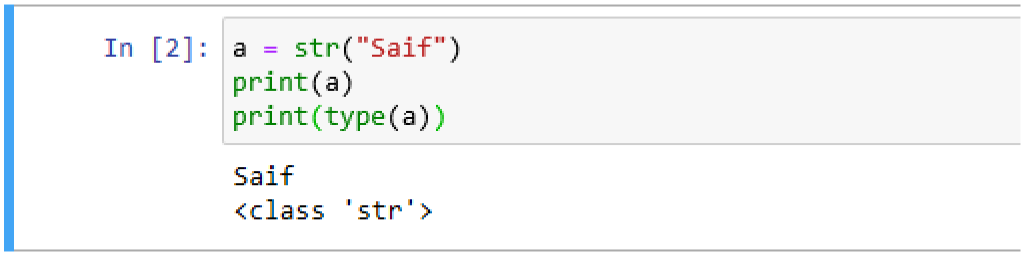
Python Strings:

In python, **strings** can be created by **enclosing** the **character** or the **sequence of characters** in the **quotes**. Python allows us to use **single** quotes, **double** quotes, or **triple** quotes to create the string.

Computers do not deal with **characters**, they deal with **numbers (binary)**. Even though you may see characters on your screen, internally it is **stored** and **manipulated** as a combination of **0s** and **1s**. This conversion of **character** to a **number** is called **encoding**, and the **reverse** process is **decoding**. **ASCII** and **Unicode** are some of the popular encodings used.

In Python, a **string** is a **sequence** of **Unicode** characters. Unicode was introduced to include every character in all languages and bring **uniformity** in **encoding**.

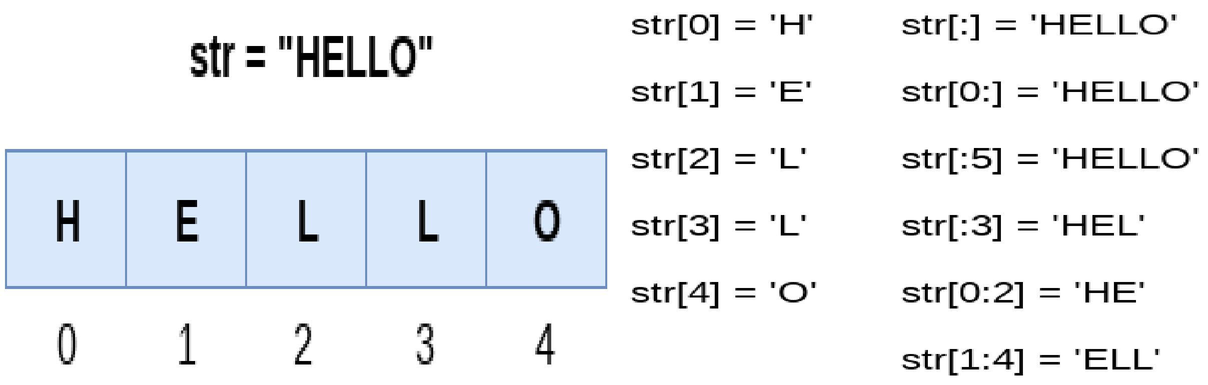
Declaring string & checking type:



```
In [2]: a = str("Saif")
        print(a)
        print(type(a))

        Saif
        <class 'str'>
```

Strings indexing and splitting:

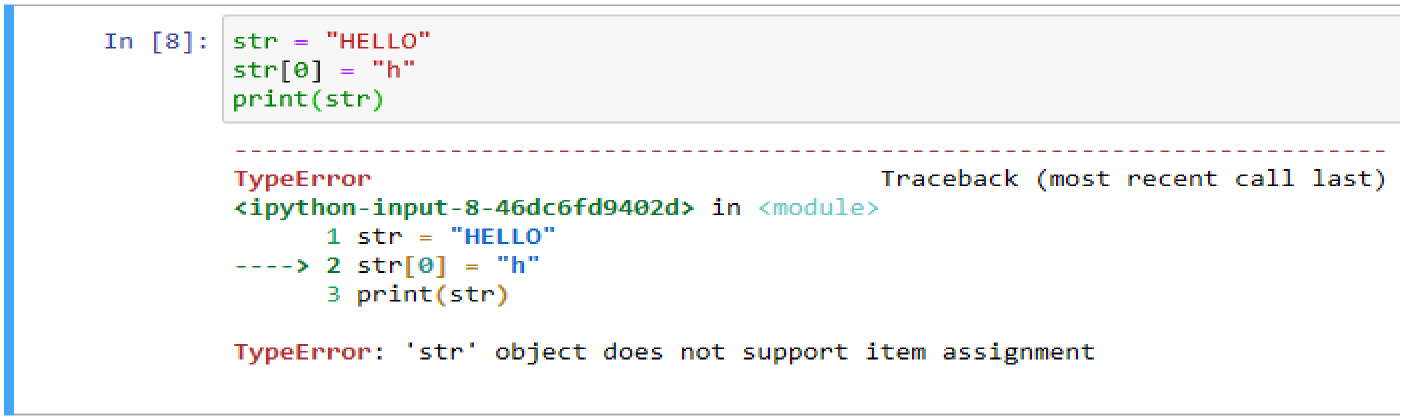
Like other languages, the **indexing** of the python strings starts from **0**. **E.g.** The string "**HELLO**" is indexed as given in the below figure.



As shown in python, the **slice operator [ ]** is used to **access** the **individual** characters of the string. However, we can use **: (colon)** operator in python to **access** the **substring**.

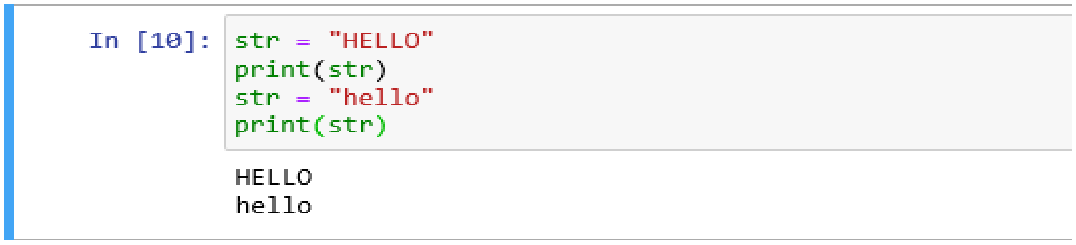
Here, we must notice that the **upper** range given in the slice operator is always **exclusive** i.e., if str = 'python' is given, then **str[1:3]** will always include **str[1] = 'p'**, **str[2] = 'y'**, **str[3] = 't'** and **nothing** else.

Reassigning (Change/Delete) strings:
Strings are **immutable**. This means that elements of a string **cannot** be changed once they have been assigned. We can simply **reassign different** strings to the **same** name.



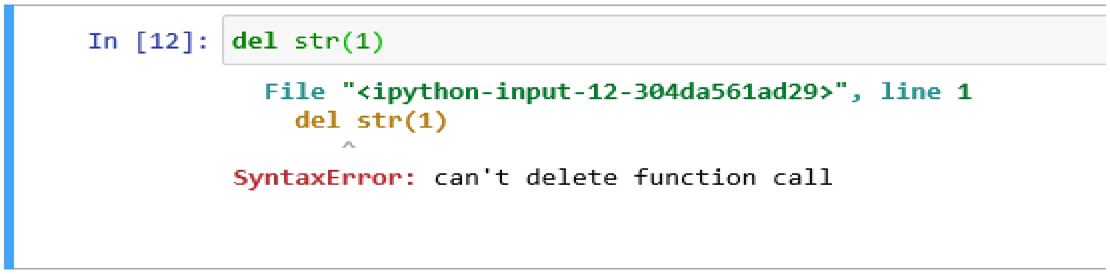
```
In [8]: str = "HELLO"
        str[0] = "h"
        print(str)

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-8-46dc6fd9402d> in <module>
              1 str = "HELLO"
        ----> 2 str[0] = "h"
              3 print(str)

        TypeError: 'str' object does not support item assignment
```

However, the string **str** can be **completely** assigned to a **new content** as specified in the following example.

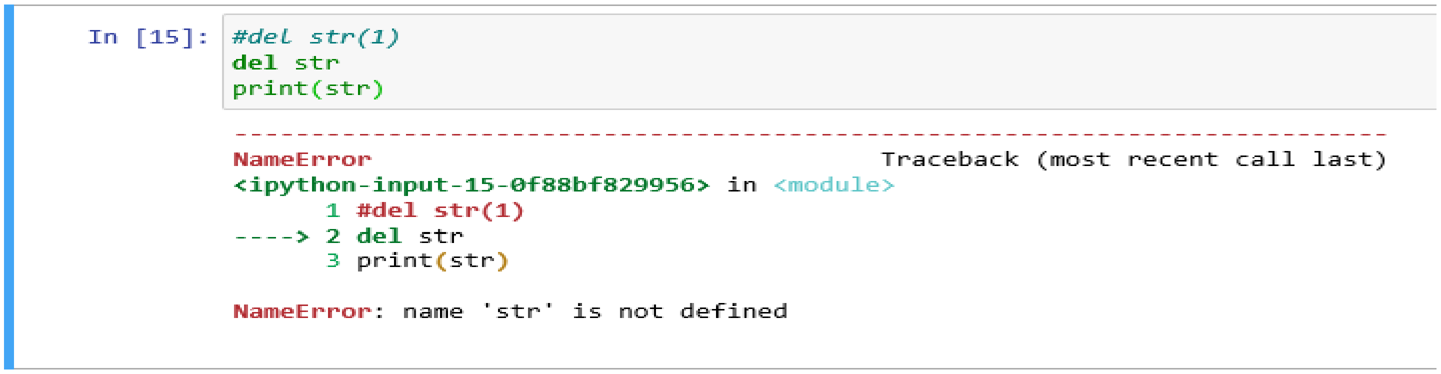


```
In [10]: str = "HELLO"
         print(str)
         str = "hello"
         print(str)

         HELLO
         hello
```

We cannot **delete** or **remove** characters from a string. But **deleting** the string **entirely** is **possible** using the **del** keyword.



```
In [12]: del str(1)

           File "<ipython-input-12-304da561ad29>", line 1
             del str(1)
                ^
         SyntaxError: can't delete function call
```

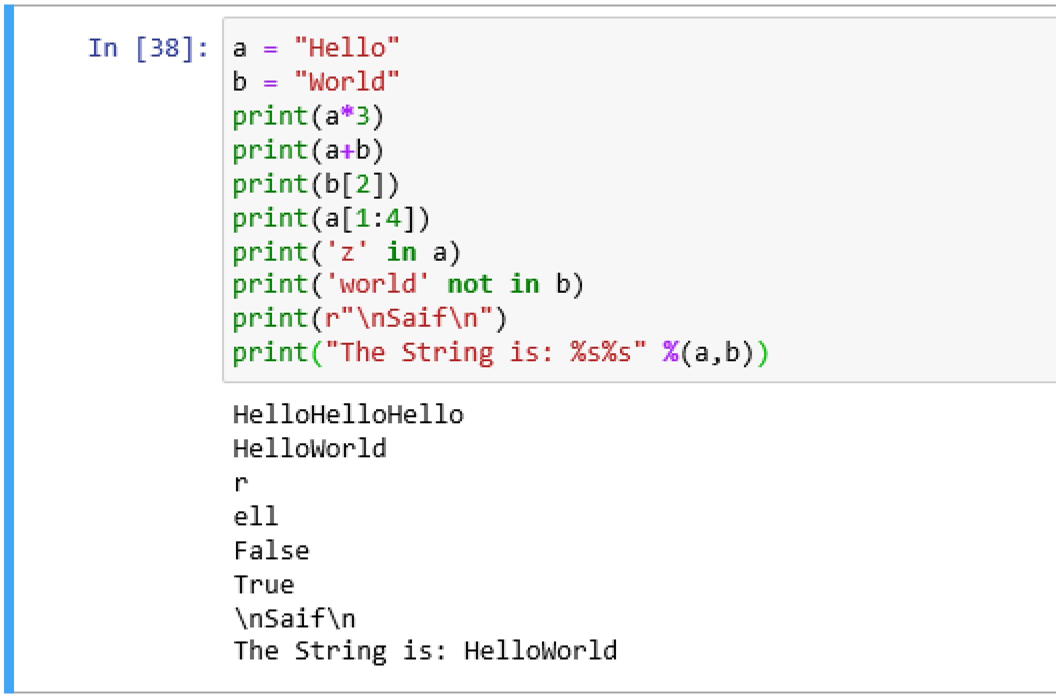


```
In [15]: #del str(1)
         del str
         print(str)

         ---------------------------------------------------------------------------
         NameError                                 Traceback (most recent call last)
         <ipython-input-15-0f88bf829956> in <module>
               1 #del str(1)
         ----> 2 del str
               3 print(str)

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

String Operators:

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[ ]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python

E.g.

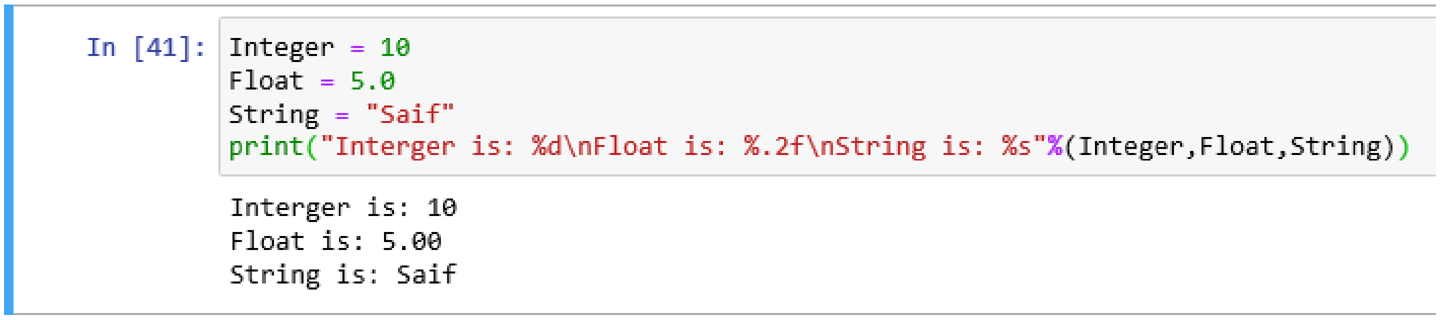


```
In [38]: a = "Hello"
         b = "World"
         print(a*3)
         print(a+b)
         print(b[2])
         print(a[1:4])
         print('z' in a)
         print('world' not in b)
         print(r"\nSaif\n")
         print("The String is: %s%s" %(a,b))
```

```
HelloHelloHello
HelloWorld
r
ell
False
True
\nSaif\n
The String is: HelloWorld
```

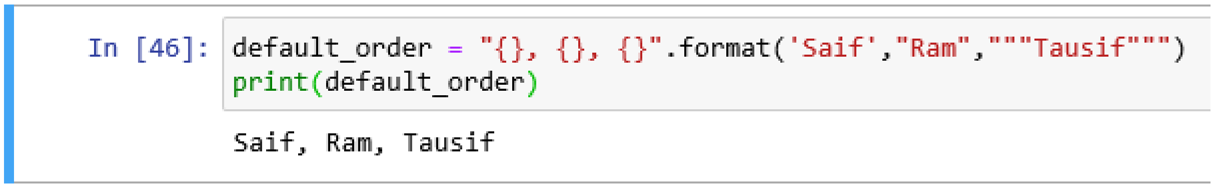
The format ( ) Method for Formatting Strings:

The **format ( )** method that is available with the string object is very versatile and powerful in formatting strings. Format strings contain **curly braces { } as placeholders** or **replacement fields** which get **replaced.** We can use **positional** or **keyword** arguments to specify the order.



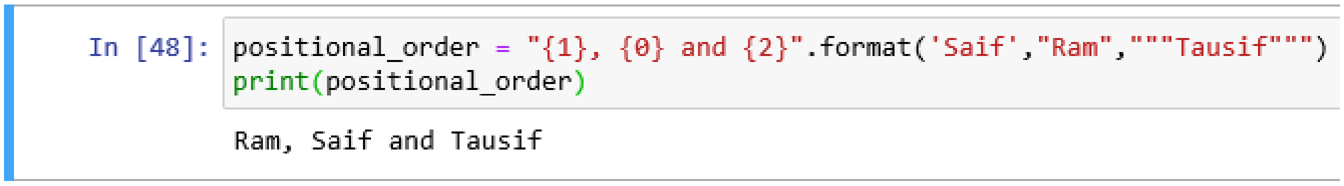
```
In [41]: Integer = 10
         Float = 5.0
         String = "Saif"
         print("Interger is: %d\nFloat is: %.2f\nString is: %s"%(Integer,Float,String))

         Interger is: 10
         Float is: 5.00
         String is: Saif
```

1) default (implicit) order:



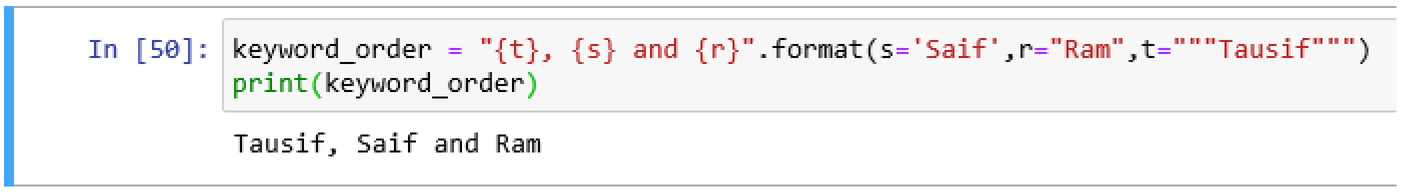
```
In [46]: default_order = "{}, {}, {}".format('Saif',"Ram","""Tausif""")
         print(default_order)

         Saif, Ram, Tausif
```

2) order using positional argument:



```
In [48]: positional_order = "{1}, {0} and {2}".format('Saif',"Ram","""Tausif""")
         print(positional_order)

         Ram, Saif and Tausif
```

3) order using keyword argument:



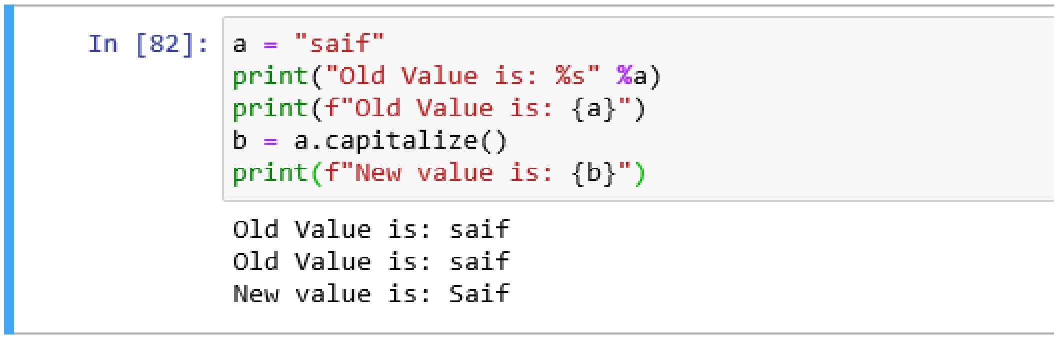
```
In [50]: keyword_order = "{t}, {s} and {r}".format(s='Saif',r="Ram",t="""Tausif""")
         print(keyword_order)

         Tausif, Saif and Ram
```

Here are some basic argument specifiers you should know:

- **%s:** String (or any object with a string representation, like numbers)
- **%d:** Integers
- **%f:** Floating point numbers
- **%.<number of digits>f:** Floating point numbers with a fixed amount of digits to the right of the dot.
- **%x/%X:** Integers in hex representation (lowercase/uppercase)

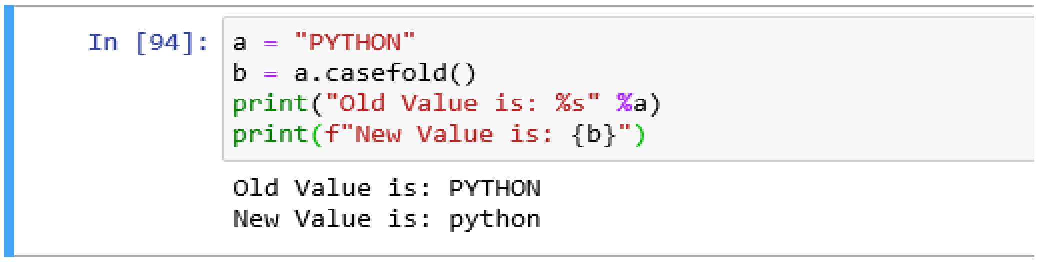
Built-in String functions:
Python provides various in-built functions that are used for string handling.

1) capitalize ( ):
Python **capitalize ( )** method converts **first** character of the string into **uppercase** without altering the whole string. It changes the **first** character only and **skips** rest of the string **unchanged**.



```
In [82]: a = "saif"
         print("Old Value is: %s" %a)
         print(f"Old Value is: {a}")
         b = a.capitalize()
         print(f"New value is: {b}")

         Old Value is: saif
         Old Value is: saif
         New value is: Saif
```

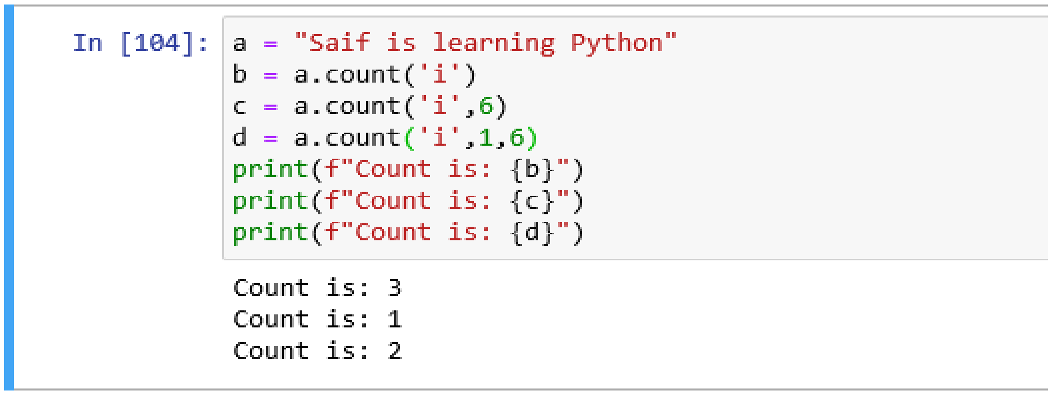
2) casefold ( ):
Python **casefold ( )** method returns a **lowercase** copy of the string. It is more similar to **lowercase** method except it removes all case distinctions present in the string.



```
In [94]: a = "PYTHON"
         b = a.casefold()
         print("Old Value is: %s" %a)
         print(f"New Value is: {b}")

         Old Value is: PYTHON
         New Value is: python
```

3) Count ( ):
It returns the **number of occurrences** of **substring** in the specified range. It takes **three** parameters, **first** is a **substring**, **second** a **start index** and **third** is **last index** of the range. **Start** and **end** both are **optional** whereas **substring** is **required**.

Syntax:
Count (sub [, start [, end]])

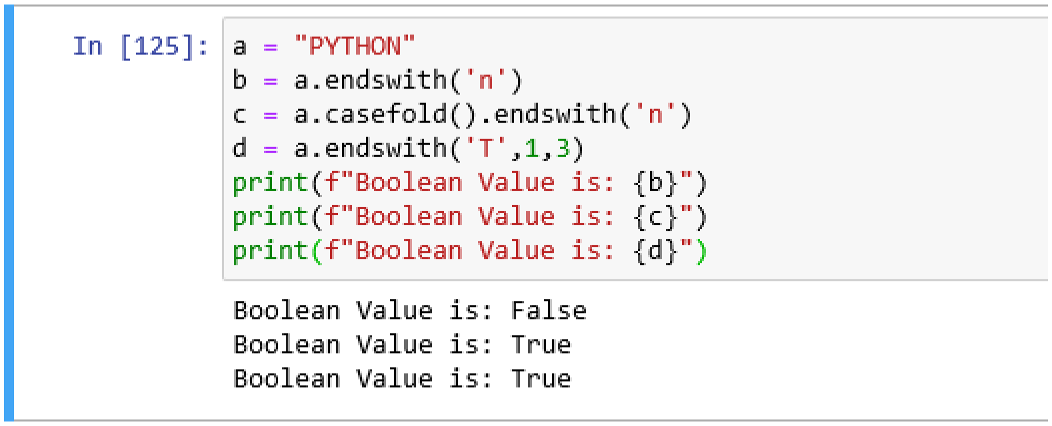


```
In [104]: a = "Saif is learning Python"
          b = a.count('i')
          c = a.count('i',6)
          d = a.count('i',1,6)
          print(f"Count is: {b}")
          print(f"Count is: {c}")
          print(f"Count is: {d}")

          Count is: 3
          Count is: 1
          Count is: 2
```

4) endswith ( ):

Python **endswith ( )** method returns **true** if the string **ends** with the specified **substring**, otherwise returns **false**.

Syntax: endswith (suffix [, start [, end]])

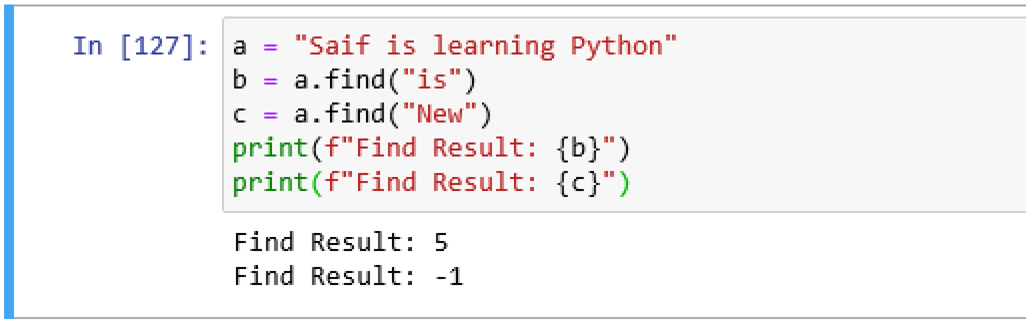


```
In [125]: a = "PYTHON"
          b = a.endswith('n')
          c = a.casefold().endswith('n')
          d = a.endswith('T',1,3)
          print(f"Boolean Value is: {b}")
          print(f"Boolean Value is: {c}")
          print(f"Boolean Value is: {d}")

          Boolean Value is: False
          Boolean Value is: True
          Boolean Value is: True
```

5) find ( ):

Python **find ( )** method finds **substring** in the **whole** string and returns **index** of the **first** match. It returns **-1** if substring **does not** match.

Syntax: find (sub [, start [, end]])

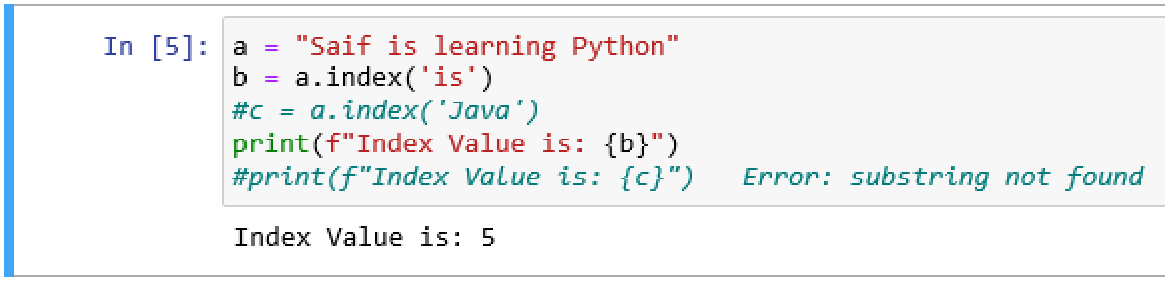


```
In [127]: a = "Saif is learning Python"
          b = a.find("is")
          c = a.find("New")
          print(f"Find Result: {b}")
          print(f"Find Result: {c}")

          Find Result: 5
          Find Result: -1
```

6) index ( ):

Python **index ( )** method is same as the **find ( )** method **except** it returns **error** on **failure**. This method returns **index** of **first** occurred substring and an **error** if there is **no match** found.

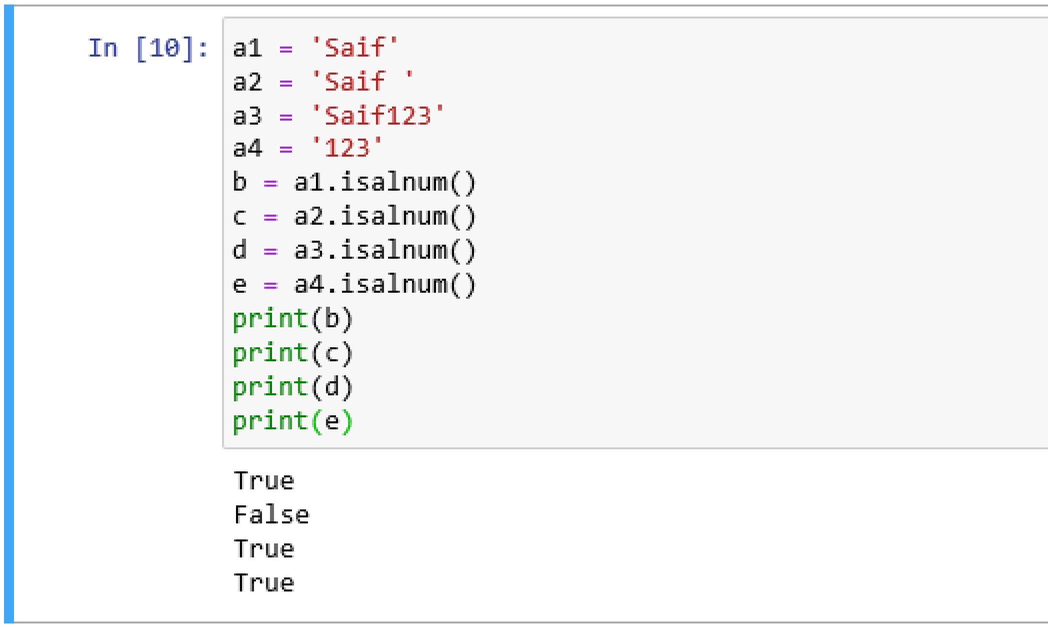
Syntax: index (sub [, start [, end]])



```
In [5]: a = "Saif is learning Python"
        b = a.index('is')
        #c = a.index('Java')
        print(f"Index Value is: {b}")
        #print(f"Index Value is: {c}")   Error: substring not found

        Index Value is: 5
```

7) isalnum ( ):

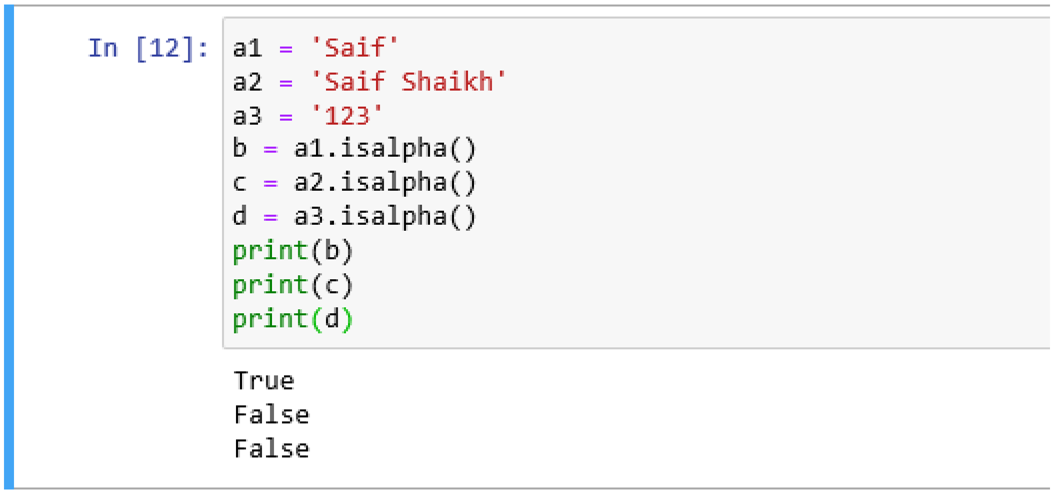
Python **isalnum ( )** method checks whether the **all characters** of the string is **alphanumeric** or **not**. A character which is either a **letter** or a **number** is known as **alphanumeric**. It **does not allow** special chars even spaces.



```
In [10]: a1 = 'Saif'
         a2 = 'Saif '
         a3 = 'Saif123'
         a4 = '123'
         b = a1.isalnum()
         c = a2.isalnum()
         d = a3.isalnum()
         e = a4.isalnum()
         print(b)
         print(c)
         print(d)
         print(e)

True
False
True
True
```

8) isalpha ( ):

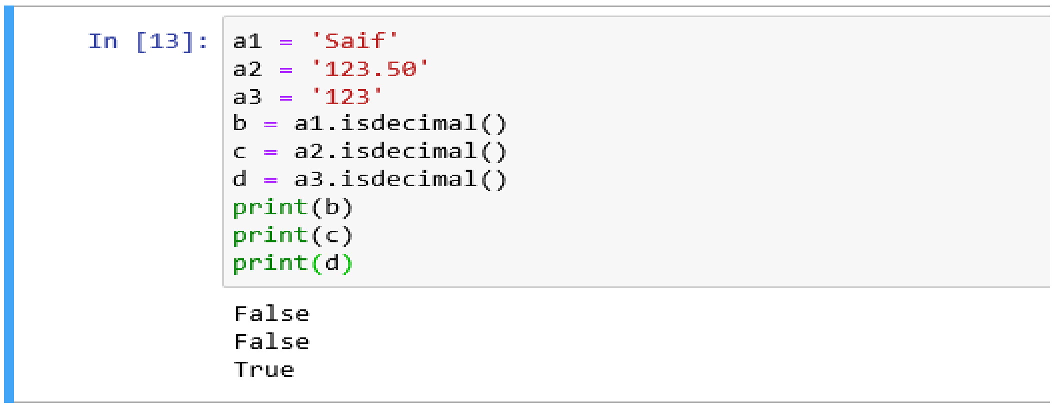
Python **isalpha ()** method returns **true** if **all characters** in the string are **alphabetic**. It returns **false** if the characters are **not alphabetic**. It returns either **True** or **False**.



```
In [12]: a1 = 'Saif'
         a2 = 'Saif Shaikh'
         a3 = '123'
         b = a1.isalpha()
         c = a2.isalpha()
         d = a3.isalpha()
         print(b)
         print(c)
         print(d)

True
False
False
```

9) isdecimal ( ):

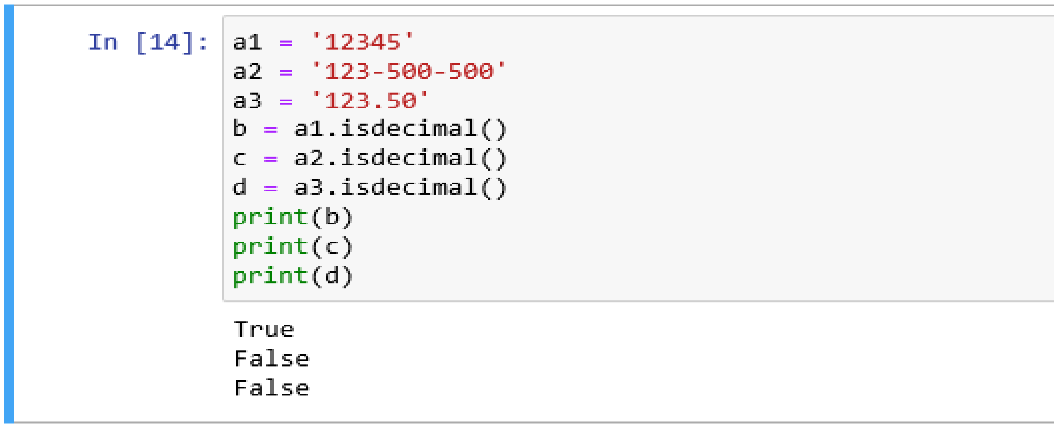
Python **isdecimal ( )** method checks whether **all the characters** in the string are **decimal** or **not**. Decimal characters are those have base 10. This method returns **Boolean** either **true** or **false**.



```
In [13]: a1 = 'Saif'
         a2 = '123.50'
         a3 = '123'
         b = a1.isdecimal()
         c = a2.isdecimal()
         d = a3.isdecimal()
         print(b)
         print(c)
         print(d)

         False
         False
         True
```

10) isdigit ( ):

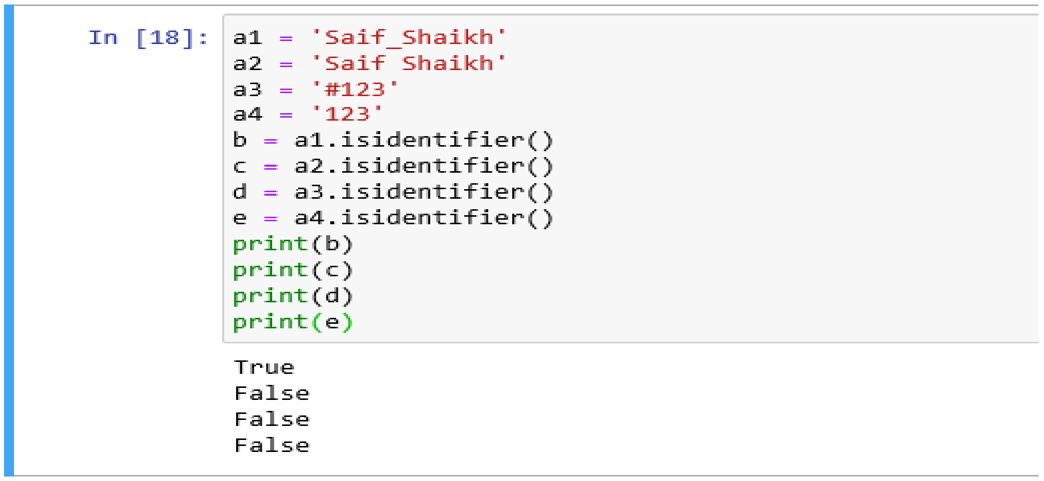
Python **isdigit ()** method returns **true** if **all the characters** in the string are **digits**. It returns **false** if **no character** is digit in the string.



```
In [14]: a1 = '12345'
         a2 = '123-500-500'
         a3 = '123.50'
         b = a1.isdecimal()
         c = a2.isdecimal()
         d = a3.isdecimal()
         print(b)
         print(c)
         print(d)

         True
         False
         False
```

11) isidentifier ( ):

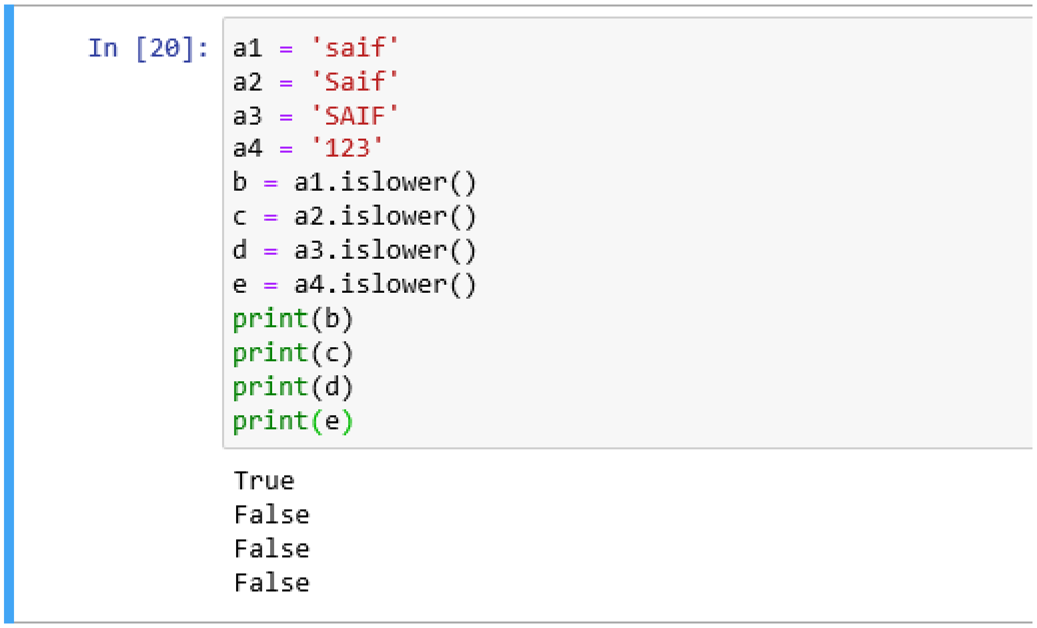
Python **isidentifier ( )** method is used to **check** whether a string is a **valid identifier** or **not**. It returns **true** if the string is a **valid identifier** otherwise returns **false**.



```
In [18]: a1 = 'Saif_Shaikh'
         a2 = 'Saif Shaikh'
         a3 = '#123'
         a4 = '123'
         b = a1.isidentifier()
         c = a2.isidentifier()
         d = a3.isidentifier()
         e = a4.isidentifier()
         print(b)
         print(c)
         print(d)
         print(e)

         True
         False
         False
         False
```

12) islower ( ):

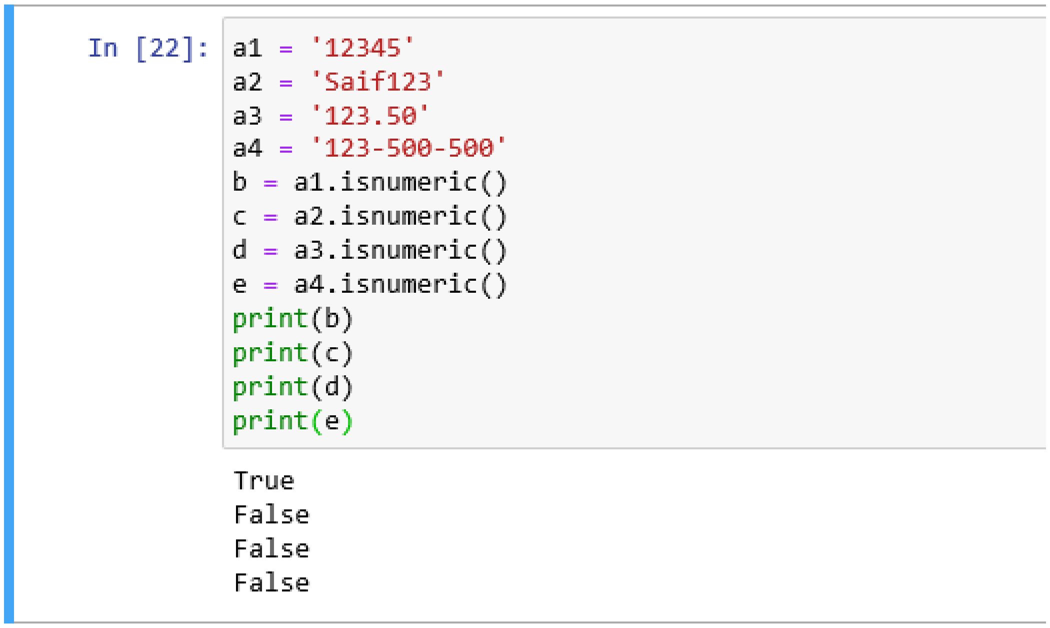
Python string **islower ( )** method returns **true** if **all characters** in the string are in **lowercase.** It returns **false** if **not** in lowercase.



```
In [20]: a1 = 'saif'
         a2 = 'Saif'
         a3 = 'SAIF'
         a4 = '123'
         b = a1.islower()
         c = a2.islower()
         d = a3.islower()
         e = a4.islower()
         print(b)
         print(c)
         print(d)
         print(e)

         True
         False
         False
         False
```

13) isnumeric ( ):

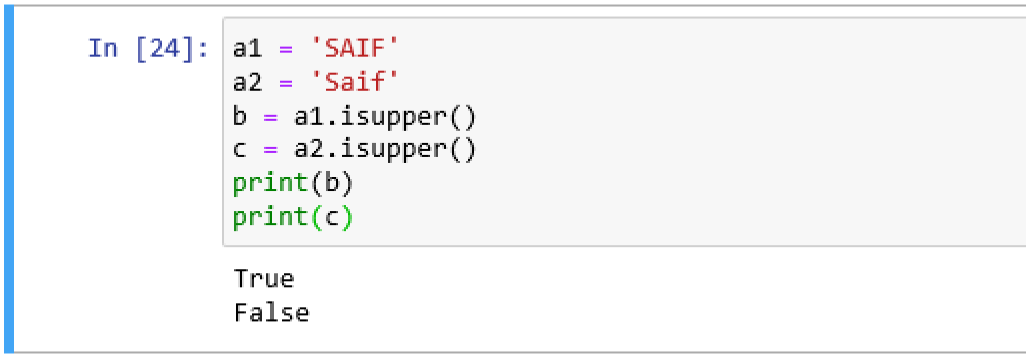
Python **isnumeric ( )** method **checks** whether **all the characters** of the string are **numeric** characters or **not.** It returns **true** if all the characters are numeric, otherwise returns **false.**



```
In [22]: a1 = '12345'
         a2 = 'Saif123'
         a3 = '123.50'
         a4 = '123-500-500'
         b = a1.isnumeric()
         c = a2.isnumeric()
         d = a3.isnumeric()
         e = a4.isnumeric()
         print(b)
         print(c)
         print(d)
         print(e)

         True
         False
         False
         False
```

14) isupper ( ):

Python **isupper ( )** method returns **true** if **all characters** in the string are in **uppercase**. It returns **false** if characters are **not** in uppercase.



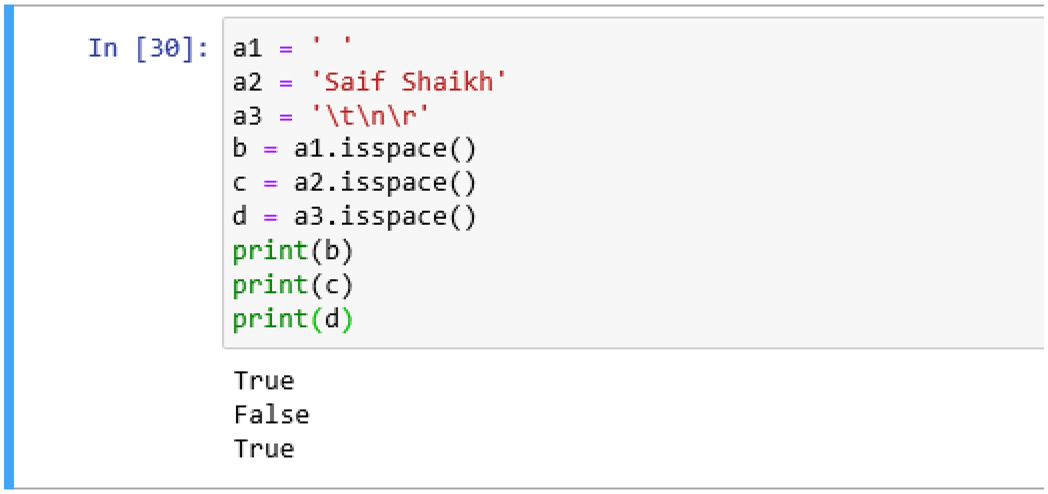
```
In [24]: a1 = 'SAIF'
         a2 = 'Saif'
         b = a1.isupper()
         c = a2.isupper()
         print(b)
         print(c)

         True
         False
```

15) isspace ( ):

Python **isspace ( )** method is used to **check space** in the string. It returns **true** if there are only **whitespace characters** in the string. Otherwise it returns **false. Space**, **newline**, and **tabs** etc. are known as **whitespace characters** and are defined in the **Unicode character** database as **other** or **Separator**.

Note: isspace ( ) method returns true for all whitespaces like:

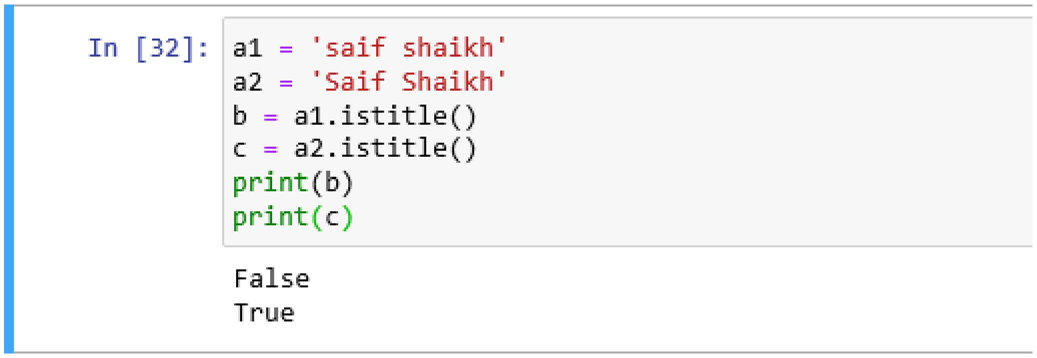
1) ' ' : Space
2) '\t': Horizontal tab
3) '\n': Newline
4) '\v': Vertical tab
5) '\f': Feed
6) '\r': Carriage return



```
In [30]: a1 = ' '
         a2 = 'Saif Shaikh'
         a3 = '\t\n\r'
         b = a1.isspace()
         c = a2.isspace()
         d = a3.isspace()
         print(b)
         print(c)
         print(d)

         True
         False
         True
```

16) istitle ( ):

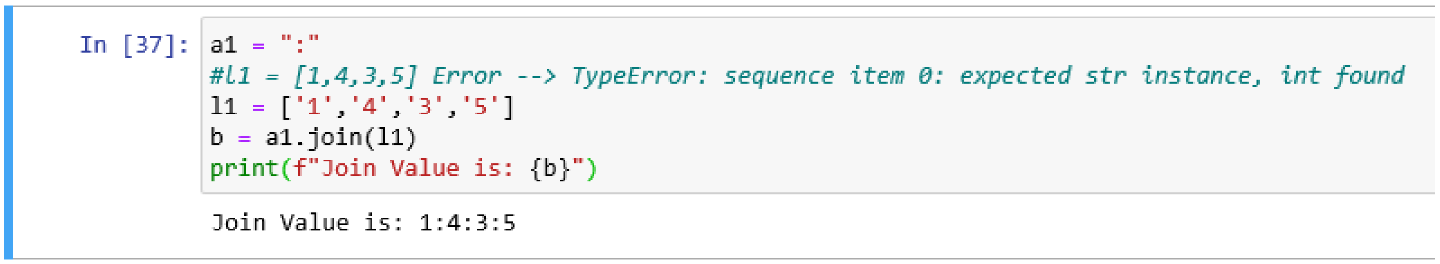
Python **istitle ( )** method returns **true** if the string is a **title cased** string. Otherwise returns **false**.



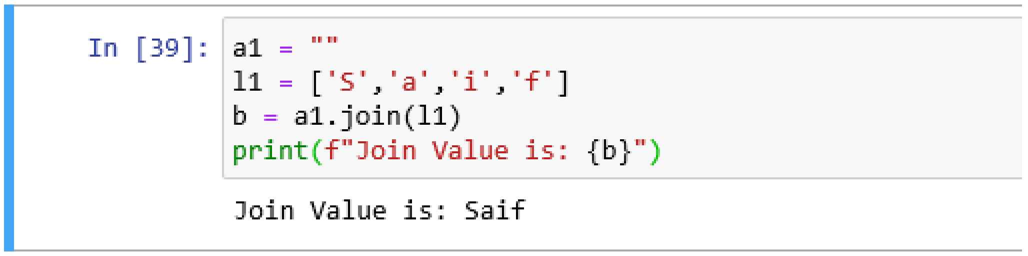
```
In [32]: a1 = 'saif shaikh'
         a2 = 'Saif Shaikh'
         b = a1.istitle()
         c = a2.istitle()
         print(b)
         print(c)

         False
         True
```

17) join ( ):

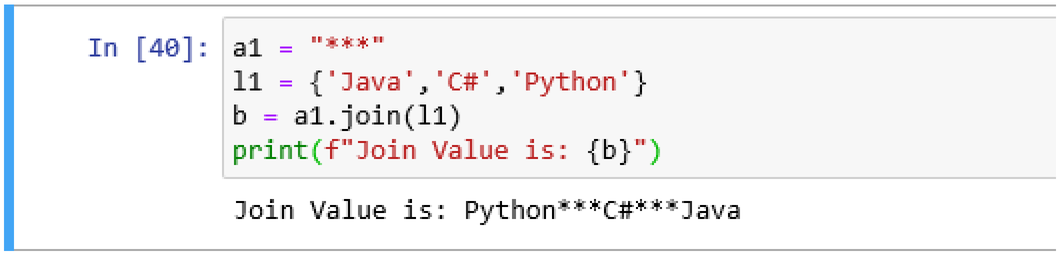
Python **join ( )** method is used to **concat** a string with **iterable object**. It returns a **new string** which is the **concatenation** of the **strings** in **iterable**. It **throws** an **exception TypeError** if iterable **contains** any **non-string** value. It **allows** various iterables like: **List**, **Tuple**, **String** etc.



```
In [37]: a1 = ":"
         #l1 = [1,4,3,5] Error --> TypeError: sequence item 0: expected str instance, int found
         l1 = ['1','4','3','5']
         b = a1.join(l1)
         print(f"Join Value is: {b}")

         Join Value is: 1:4:3:5
```

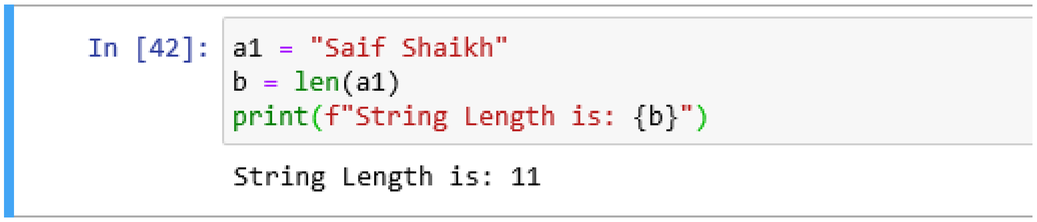


```
In [39]: a1 = ""
         l1 = ['S','a','i','f']
         b = a1.join(l1)
         print(f"Join Value is: {b}")

         Join Value is: Saif
```



```
In [40]: a1 = "***"
         l1 = {'Java','C#','Python'}
         b = a1.join(l1)
         print(f"Join Value is: {b}")

         Join Value is: Python***C#***Java
```

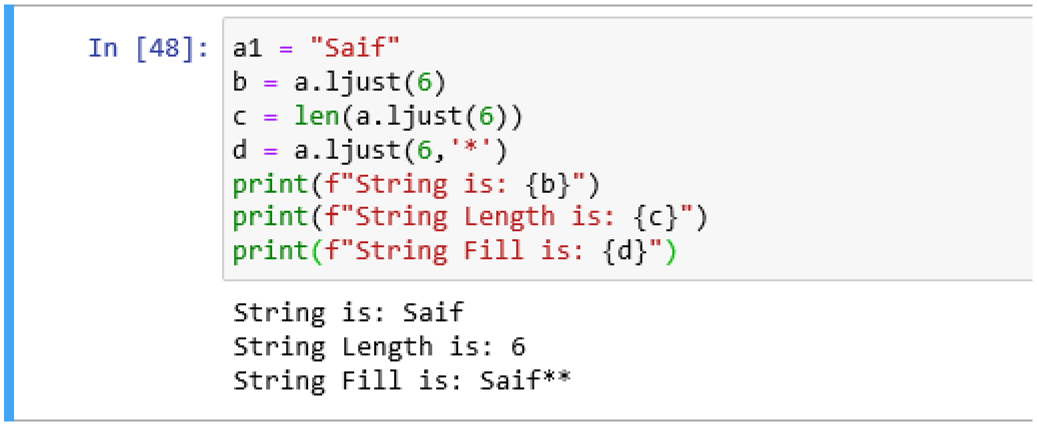
18) len ( ): It returns the **length** of a string.



```
In [42]: a1 = "Saif Shaikh"
         b = len(a1)
         print(f"String Length is: {b}")

         String Length is: 11
```

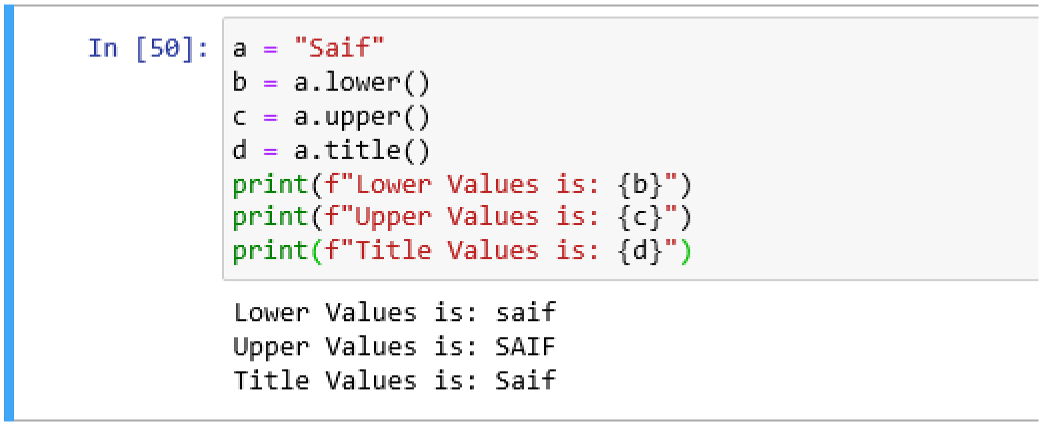
19) ljust ( ):
Python **ljust ( )** method **left justify** the string and **fill** the **remaining spaces** with **fillchars**. This method returns a new string **justified left** and **filled** with **fillchars**.

Syntax:
ljust (width[, fillchar])



```
In [48]: a1 = "Saif"
         b = a.ljust(6)
         c = len(a.ljust(6))
         d = a.ljust(6,'*')
         print(f"String is: {b}")
         print(f"String Length is: {c}")
         print(f"String Fill is: {d}")

         String is: Saif
         String Length is: 6
         String Fill is: Saif**
```

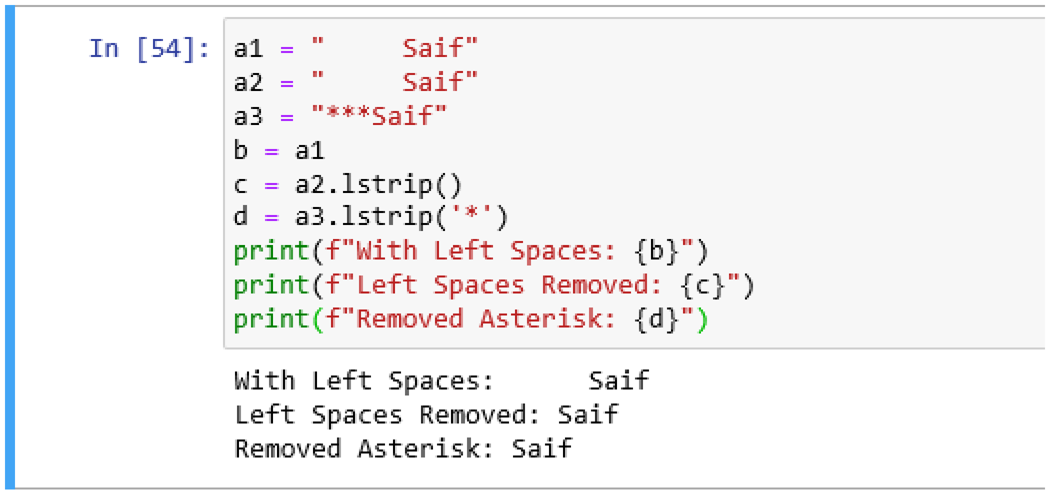
20) lower ( ) & upper ( ):
Python **lower ( )**, **upper ( )** method returns a copy of the string after converting all the characters into lowercase and **uppercase** respectively.



```
In [50]: a = "Saif"
         b = a.lower()
         c = a.upper()
         d = a.title()
         print(f"Lower Values is: {b}")
         print(f"Upper Values is: {c}")
         print(f"Title Values is: {d}")

         Lower Values is: saif
         Upper Values is: SAIF
         Title Values is: Saif
```

21) lstrip ( ):

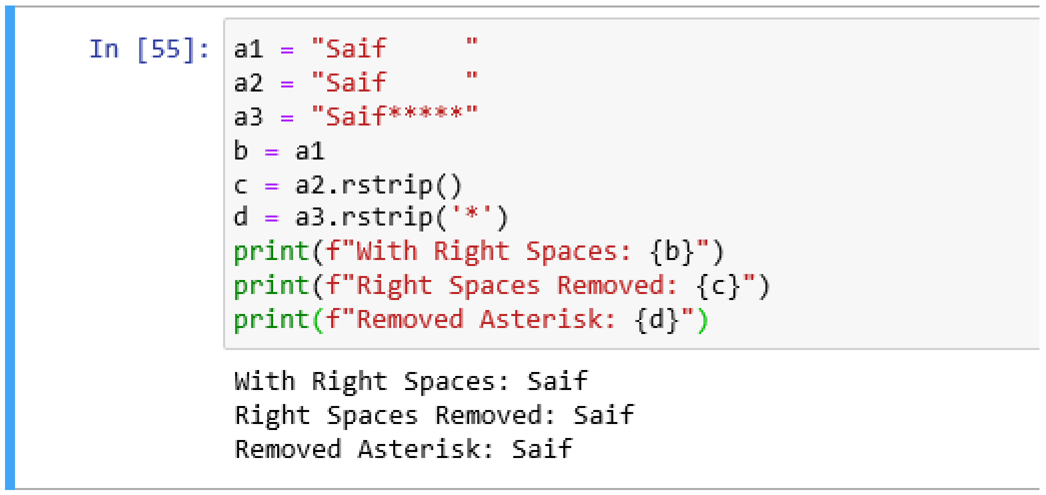
Python **lstrip ( )** method is used to **remove all leading characters** from the string. It takes a **char type parameter** which is **optional**. If parameter is **not** provided, it **removes** all the **leading spaces** from the string.



```
In [54]: a1 = "     Saif"
         a2 = "     Saif"
         a3 = "***Saif"
         b = a1
         c = a2.lstrip()
         d = a3.lstrip('*')
         print(f"With Left Spaces: {b}")
         print(f"Left Spaces Removed: {c}")
         print(f"Removed Asterisk: {d}")

         With Left Spaces:      Saif
         Left Spaces Removed: Saif
         Removed Asterisk: Saif
```

22) rstrip ( ):

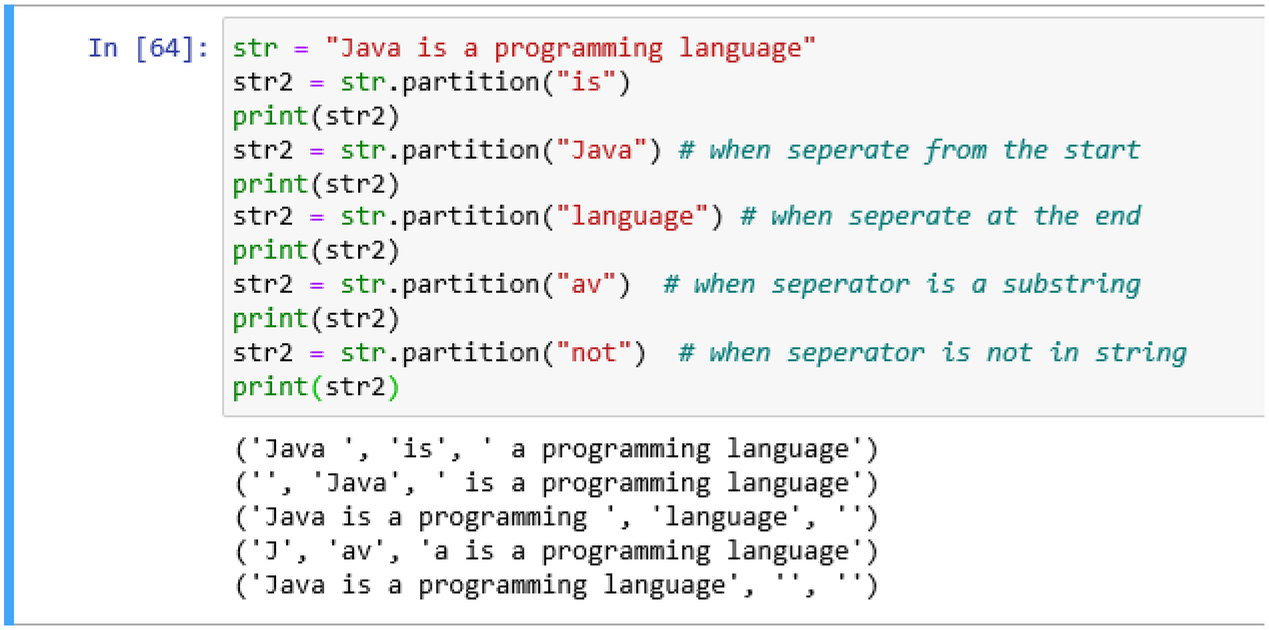
Python **rstrip ( )** method **removes all the trailing characters** from the string. It means it **removes** all the **specified characters** from **right side** of the string. If we **don't** specify the parameter, it **removes** all the **whitespaces** from the string. This method returns a **string** value.



```
In [55]: a1 = "Saif     "
         a2 = "Saif     "
         a3 = "Saif*****"
         b = a1
         c = a2.rstrip()
         d = a3.rstrip('*')
         print(f"With Right Spaces: {b}")
         print(f"Right Spaces Removed: {c}")
         print(f"Removed Asterisk: {d}")

         With Right Spaces: Saif
         Right Spaces Removed: Saif
         Removed Asterisk: Saif
```

23) partition ( ):

Python **partition ( )** method **splits** the string from the string specified in parameter. It **splits** the string from at the **first occurrence** of ***parameter*** and returns a **tuple.** The tuple contains the **three** parts **before** the **separator**, the **separator itself**, and the part after the **separator**. If the **separator** is **not** found, it returns a **tuple** containing **string itself** and **two empty strings**.



```
In [64]: str = "Java is a programming language"
         str2 = str.partition("is")
         print(str2)
         str2 = str.partition("Java") # when seperate from the start
         print(str2)
         str2 = str.partition("language") # when seperate at the end
         print(str2)
         str2 = str.partition("av")  # when seperator is a substring
         print(str2)
         str2 = str.partition("not")  # when seperator is not in string
         print(str2)
```

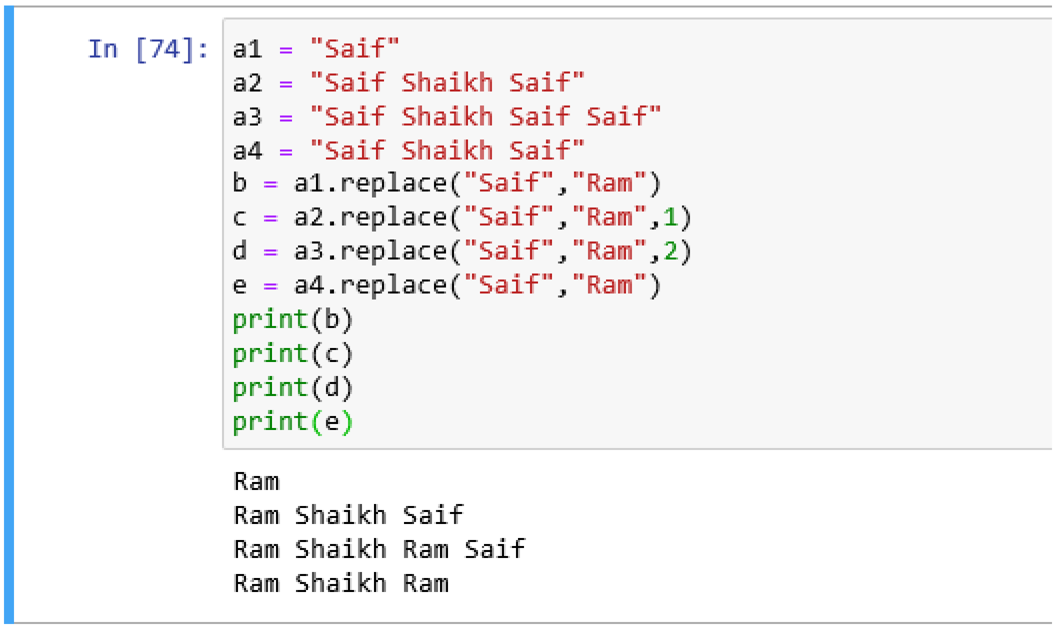
```
('Java ', 'is', ' a programming language')
('', 'Java', ' is a programming language')
('Java is a programming ', 'language', '')
('J', 'av', 'a is a programming language')
('Java is a programming language', '', '')
```

24) replace ( ):

Return a copy of the string with **all occurrences** of substring ***old*** replaced by ***new***. If the optional argument ***count*** is given, only the **first *count*** occurrences are **replaced**.

Syntax:

replace (old, new [, count])



```
In [74]: a1 = "Saif"
         a2 = "Saif Shaikh Saif"
         a3 = "Saif Shaikh Saif Saif"
         a4 = "Saif Shaikh Saif"
         b = a1.replace("Saif","Ram")
         c = a2.replace("Saif","Ram",1)
         d = a3.replace("Saif","Ram",2)
         e = a4.replace("Saif","Ram")
         print(b)
         print(c)
         print(d)
         print(e)
```

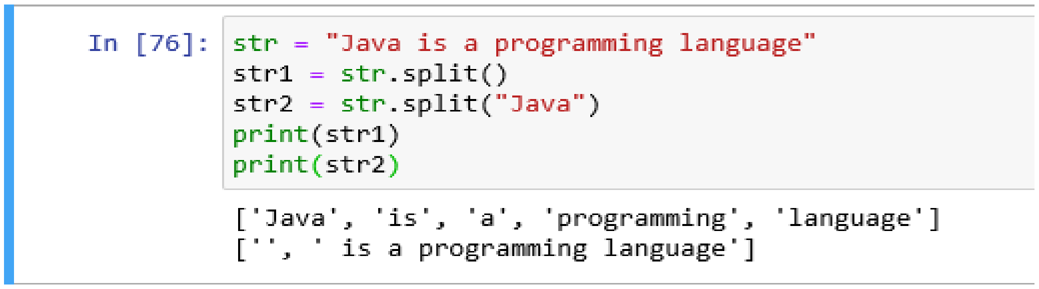
```
Ram
Ram Shaikh Saif
Ram Shaikh Ram Saif
Ram Shaikh Ram
```

25) split ( ):

Python **split ( )** method **splits** the string into a **comma separated list**. It separates string based on the **separator delimiter**. This method takes **two parameters** and **both** are **optional.** It is described below.

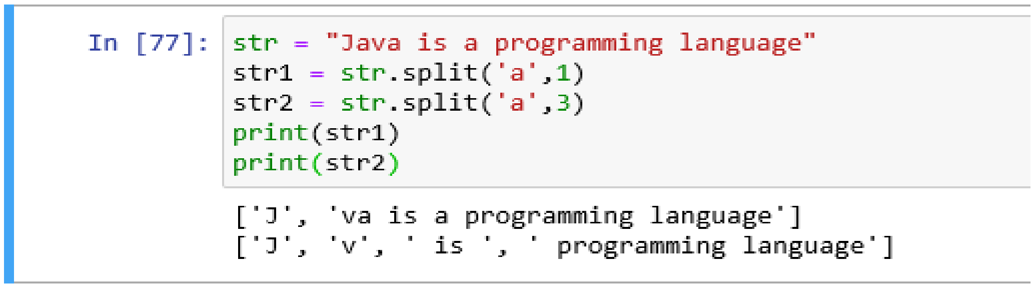
Syntax:

split (sep=None, maxsplit=-1)



```
In [76]: str = "Java is a programming language"
         str1 = str.split()
         str2 = str.split("Java")
         print(str1)
         print(str2)

         ['Java', 'is', 'a', 'programming', 'language']
         ['', ' is a programming language']
```

Along with **separator**, we can also pass **maxsplit** value. The **maxsplit** is used to **set** the **number of times** to **split.**



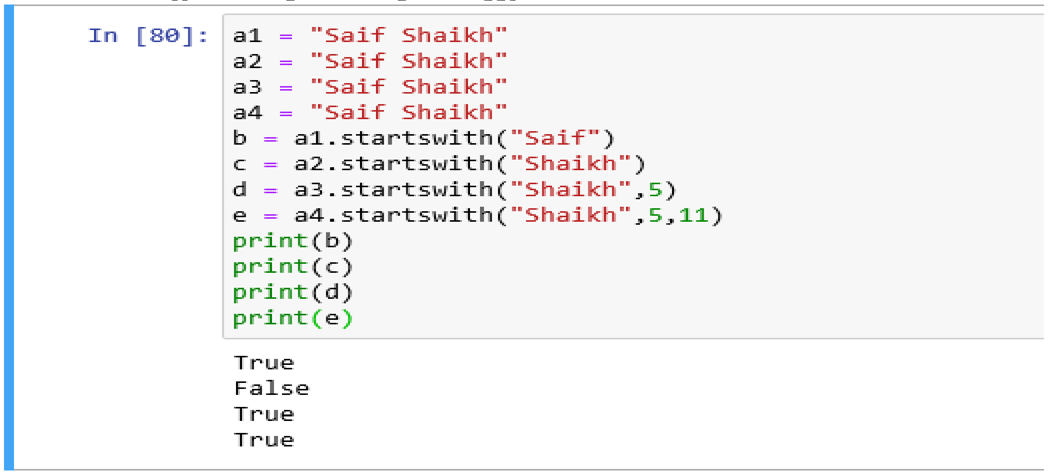
```
In [77]: str = "Java is a programming language"
         str1 = str.split('a',1)
         str2 = str.split('a',3)
         print(str1)
         print(str2)

         ['J', 'va is a programming language']
         ['J', 'v', ' is ', ' programming language']
```

26) startswith ( ):

Python **startswith ()** method returns either **true** or **false**. It returns **true** if the string **starts** with the **prefix**, otherwise **false**. It takes **two** parameters **start** and **end. Start** is a **starting index** from where **searching starts** and **end index** is where **searching stops.**

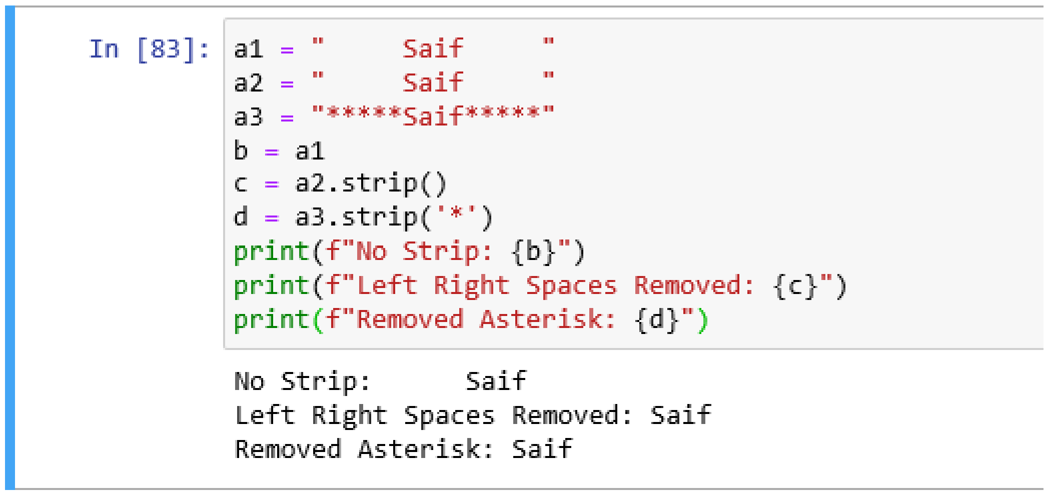
Syntax:

startswith (prefix [, start [, end]])



```
In [80]: a1 = "Saif Shaikh"
         a2 = "Saif Shaikh"
         a3 = "Saif Shaikh"
         a4 = "Saif Shaikh"
         b = a1.startswith("Saif")
         c = a2.startswith("Shaikh")
         d = a3.startswith("Shaikh",5)
         e = a4.startswith("Shaikh",5,11)
         print(b)
         print(c)
         print(d)
         print(e)

         True
         False
         True
         True
```

27) strip ([chars]): It is used to perform **lstrip ( )** and **rstrip ( )** on the string.



In [83]:
```
a1 = "     Saif     "
a2 = "     Saif     "
a3 = "*****Saif*****"
b = a1
c = a2.strip()
d = a3.strip('*')
print(f"No Strip: {b}")
print(f"Left Right Spaces Removed: {c}")
print(f"Removed Asterisk: {d}")
```

```
No Strip:      Saif
Left Right Spaces Removed: Saif
Removed Asterisk: Saif
```