# Python Programming

Python is a powerful multipurpose programming language created by Guido van Rossum.

It has a simple and easy-to-use syntax, making it a popular first-choice programming language for beginners.

## What is Python Programming Language?

Python is an interpreted, object-oriented, high-level programming language. As it is general-purpose, it has a wide range of applications from web development, building desktop GUI to scientific and mathematical computing.

## Features

- Simple and easy to learn
- Free and open-source
- Portability
- Extensible and Embeddable
- High-Level Interpreted Language
- Rich library and large community

## Reasons to Choose Python as First Language

1. Simple Elegant Syntax

```
a = 2
b = 3
sum = a + b
print(sum)
```

Even if you have never programmed before, you can easily guess that this program adds two numbers and displays it.

1. Not overly strict: You don't need to define the type of a `variable` in Python. Also, it's not necessary to add a semicolon `;` at the end of the statement.

2. The expressiveness of the language: greater functionality with fewer lines of code.

```
a = 15
b = 27
print(f'Before swapping: a, b = {a},{b}')
a, b = b, a
print(f'After swapping: a, b = {a},{b}')
```

## Applications of Python

1. Web Applications
2. AI & Machine Learning
3. Desktop GUI Applications
4. Software Development
5. Scientific and Numeric
6. Business Applications
7. Console Based Application
8. Audio or Video based Applications
9. 3D CAD Applications
10. Enterprise Applications
11. Applications for Images
12. Games and 3D Graphics

# Python Installation

I recommend reading the documentation from https://www.python.org/ OR https://www.anaconda.com/ to fully install Phyton locally on your system. Good thing is, Python is available on all -- Windows, Mac, and Linux!

# Hello, World!

Now that we have Python up and running, we can write our first Python program.

Let's create a very simple program called `Hello, World!`. A **"Hello, World!"** is a simple program that outputs Hello, World! on the screen. Since it's a very simple program, it's often used to introduce a new programming language to beginners.

`print("Hello, World!")`

**Congratulations!** You just wrote your first program in Python.

## Let's breakdown the components of this code.

1. `print()` is a function that tells the computer to perform an action. We know it is a function because it uses parentheses. print() tells Python to display or output whatever we put in the parentheses. By default, this will output to the current terminal window.
2. Some functions, like the `print()` function, are built-in functions included in Python by default. These built-in functions are always available for us to use in programs that we create. We can also define our own functions that we construct ourselves through other elements.
3. Inside the parentheses of the `print()` function is a sequence of characters — **Hello, World!** — that is enclosed in quotation marks `'` or `"`. Any characters that are inside of quotation marks are called a **string**.

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

```
Hello, World!
```

**Congratulations!** You have written the "Hello, World!" program in *Python 3*. Since the program ran, you can now confirm that Python 3 is properly installed and that the program is syntactically correct.

## How to print blank lines

Sometimes you need to print one blank line in your Python program.

```
print ("Hello World!")
print (9 * "\n")
# print ("\n\n\n\n\n\n\n\n\n")
print ("Hello World!")
```

```
Hello World!









Hello World!
```

```
print ("Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug\nSep\nOct\nNov\nDec")
```

```
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
```

```
print ("I want \\n to be printed.")
print("I'm very *happy*")
```

```
I want \n to be printed.
I'm very *happy*
```

```
print ("Hello\tWorld!") # \t is equal to 4 spaces
```

```
Hello World!
```

```
print ("""
Routine:
\t- Eat
\t- Study
\t- RoboAI
\t- Sleep\n\t- Repeat
""")
```

```
Routine:
      - Eat
      - Study
      - RoboAI
      - Sleep
      - Repeat
```

## `print()` **end** command

By default, python's `print()` function ends with a newline. This function comes with a parameter called `end`. The default value of this parameter is `\n`, i.e., the new line character. You can end a print statement with any character or string using this parameter.

```
print ("Welcome to", end = ' ')
print ("Python", end = '!')
```

```
Welcome to Python!
```

```
print("Python " , end = '@') # ends the output with '@'.
```

```
Python @
```

```
print ('_M'*9)
```

```
_M_M_M_M_M_M_M_M_M
```

```
print("*knock knock knock* Penny! \n" * 3)
# TBBT reference - iykyk!
```

```
*knock knock knock* Penny!
*knock knock knock* Penny!
*knock knock knock* Penny!
```

# Data-types

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

## 1. Numbers

Integers, floating point numbers and complex numbers fall under **Python numbers** category. They are defined as `int`, `float` and `complex` classes.

We can use the `type()` function to know which class a variable or a value belongs to.

```
a = 6
print(a, "is of type", type(a))
```

```
6 is of type <class 'int'>
```

Similarly, the `isinstance()` function is used to check if an object belongs to a particular class.

```
print(a, "is integer number?", isinstance(a,int)) # answers in
True/False
```

```
6 is integer number? True
```

```
a = 3.0
print(a, "is of type", type(a))
print(a, "is float number?", isinstance(a,float))
```

```
3.0 is of type <class 'float'>
3.0 is float number? True
```

```
a = 1+2j  # '1' is real part and '2j' is imaginary part
print(a, "is of type", type(a))
print(a, "is complex number?", isinstance(a,complex))
```

```
(1+2j) is of type <class 'complex'>
(1+2j) is complex number? True
```

```
a = 12345678901234567 89
print (a)

b = 0.12345678901234567 89  # total of only 17 numbers after decimal
can be printed.
print (b)

c = 1+2j
print (c)
```

```
1234567890123456789
0.12345678901234568
(1+2j)
```

## 2. List `[]`

List is an **ordered sequence** of items.

```
x = [6, 99, 77, 'Apple']
print(x, "is of type", type(x))
```

```
[6, 99, 77, 'Apple'] is of type <class 'list'>
```

```
a = [5, 10, 15, 20, 25, 30, 35, 40]  # Total elemnets is 8
#   [0   1   2   3   4   5   6   7]  ➡ Index forward
#   [-8 -7  -6  -5  -4  -3  -2  -1]  ⏩ Index backward

# index '0' is element '1' = 5,
# index '1' is element '2' = 10,
# index '2' is element '3' = 15,
# .
# .
# .
# index '7' is element '8' = 40,

a[1] # To access the elements in the list

# a[2] = 15
print("a[2] = ", a[2])

# a[0:3] = [5, 10, 15]
print("a[0:3] = ", a[0:3])  # [0:3] means elements from 0 uptil 2
index (not include last element)
                            # [0:3] means from index 0 to 3 - 1
                            # [0:3] means from index 0 to 2

# a[5:] = [30, 35, 40]  # [5:] means all the elements from 5 till end
print("a[5:] = ", a[5:])
```

```
a[2] =  15
a[0:3] =  [5, 10, 15]
a[5:] =  [30, 35, 40]
```

```
a[1:-2]
```

```
[10, 15, 20, 25, 30]
```

```
a[5:9]
```

```
[30, 35, 40]
```

```
a[:5]
```

```
[5, 10, 15, 20, 25]
```

```
# Change the element of the List
a = [1, 2, 3]
#   [0  1  2] ⇨ Index forward

a[2] = 4  # Change my third element from '3' to '4' # [2] is the index
number
print(a)
```

```
[1, 2, 4]
```

## 3. Tuple `()`

Tuple is an **ordered sequence** of items same as a list. The only difference is that tuples are **immutable**. *Tuples once created cannot be modified.*

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.

```
# Tuple 't' have 3 elements
t = (6,'program', 1+3j)
#   (0        1      2) ⇨ Index forward

# index '0' is element '1'= 6
# index '1' is element '2'= program
# index '2' is elemtnt '3'= 1+3j

# t[1] = 'program'
print("t[1] = ", t[1])

# t[0:3] = (6, 'program', (1+3j))
print("t[0:3] = ", t[0:3])

# Generates error
# Tuples are immutable
t[0] = 10  # trying to change element 0 from '6' to '10'
```

```
t[1] =  program
t[0:3] =  (6, 'program', (1+3j))
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-26-c92d9f2b36de> in <cell line: 17>()
     15 # Generates error
```

```
     16 # Tuples are immutable
---> 17 t[0] = 10  # trying to change element 0 from '6' to '10'

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

```
list1 =  [9, 'apple', 3 + 6j]  # list
tuple1 = (9, 'apple', 3 + 6j)  # tuple

list1[1] = 'banana'  # List is mutable
print(list1)         # No error
tuple1[1]= 'banana'  # Tuple is immutable
print(tuple1)        # error
```

```
[9, 'banana', (3+6j)]
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-27-e32e417070a1> in <cell line: 6>()
      4 list1[1] = 'banana'  # List is mutable
      5 print(list1)         # No error
----> 6 tuple1[1]= 'banana'  # Tuple is immutable
      7 print(tuple1)        # error

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

# 4. Strings

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, `'''` or `"""`.

```
s = '''Apple'''
print(s)
s = """Apple"""
print(s)
s = 'Apple'
print(s)
s = "Apple"
print(s)
s = Apple   # cannot write string with out quotes ('', " ", """ """,
''' ''')
print(s)
```

```
Apple
Apple
Apple
Apple
```

```
---------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
<ipython-input-28-7fce570bf337> in <cell line: 9>()
      7 s = "Apple"
      8 print(s)
----> 9 s = Apple   # cannot write string with out quotes ('', " ",
""" """, ''' ''')
     10 print(s)

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

```
s = "This is a string"  # s is my variable
print(s)
s = '''A multiline
string'''
print(s)
```

```
This is a string
A multiline
string
```

Just like a list and tuple, the slicing operator `[ ]` can be used with strings. Strings, however, are **immutable**.

```
s = 'Hello world!' # total 12 elements. Index start from '0' to '11'

# s[4] = 'o'
print("s[4] = ", s[4])

# s[6:11] = 'world' # index '6' to '11' means element from 6 to 10
print("s[6:11] = ", s[6:11])
```

```
s[4] =  o
s[6:11] =  world
```

```
a = "apple"

a[0]='o'

# Simiar to TUPLE, STRING is immutable
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-31-466b90e7ef2f> in <cell line: 3>()
      1 a = "apple"
      2
```

```
----> 3 a[0]='o'
      4
      5 # Simiar to TUPLE, STRING is immutable

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

## 5. Set `{ }`

Set is an **unordered collection** of unique items. Set is defined by values separated by comma inside braces `{ }`. Items in a set are not ordered.

```
a = {7,1,3,6,9}

# printing set variable
print("a = ", a)

# data type of variable a
print(type(a))

a =  {1, 3, 6, 7, 9}
<class 'set'>
```

We can perform set operations like union, intersection on two sets. Sets have unique values. They eliminate duplicates.

```
a = {1,2,2,3,3,3} # we can see total 6 elements
print(a)

{1, 2, 3}
```

Since, set are unordered collection, indexing has no meaning. Hence, the slicing operator `[]` does not work.

```
a = {1,2,3}  # in Set data type we cannot access the elements because
set is unordered collection
a[1]  # Index [1] means element 2

---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-34-242b77ef2a87> in <cell line: 2>()
      1 a = {1,2,3}  # in Set data type we cannot access the elements
because set is unordered collection
----> 2 a[1]  # Index [1] means element 2

TypeError: 'set' object is not subscriptable
```

## 6. Dictionary `{:}`

Dictionary is an **unordered collection** of **key-value pairs**. It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value. 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.

```
d = {1: 'Apple', 2: 'Cat', 3: 'Food'}  # 'Apple' is element and 1 is the key of element.
print(d, type(d))

d[3]
```

```
{1: 'Apple', 2: 'Cat', 3: 'Food'} <class 'dict'>
```

```
{"type":"string"}
```

```
d = {1:'value','key':2} # d is my variable, 'value' and 'key' are the element and 1 and 2 are the key.
type(d)
```

```
dict
```

```
d = {1:'value','key':2} # '1' is the key to access 'value' and 'key' is the key to access '2'
print(type(d))

print("d[1] = ", d[1]); # try to find the element from key.

print("d['key'] = ", d['key']);  # try to find the key from the element.
```

```
<class 'dict'>
d[1] =  value
d['key'] =  2
```

```
print(type(zip([1,2],[3,4])))
```

```
<class 'zip'>
```

# Operators

Python is also a calculator, lol.

Q.> What are **Operators**? Operators are special **symbols** in Python that carry out **arithmetic** or **logical computation**. The value that the operator operates on is called the **operand**.

```
6+3
# Here, + is the operator that performs addition.
# 6 and 3 are the operands and 9 is the output of the operation.
```

```
9
```

# 1. Arithmetic

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

| Symbol | Task Performed | Meaning | Example |
|---|---|---|---|
| `+` | Addition | add two operands or unary plus | **x + y** or **+2** |
| `-` | Subtraction | substract right operand from the left or unary minus | **x - y** or **-2** |
| `*` | Multiplication | Multiply two operands | **x * y** |
| `/` | Division | Divide left operand by the right one (always results into float) | **x / y** |
| `%` | Modulus (remainder) | remainder of the division of left operand by the right | **x % y** (remainder of **x/y**) |
| `//` | Integer/Floor division | division that results into whole number adjusted to the left in the number line | **x // y** |
| `**` | Exponentiation (power) | left operand raised to the power of right | **x ** y** (**x** to the power **y**) |

```
print('Addition: ', 1 + 2)
print('Subtraction: ', 2 - 1)
print('Multiplication: ', 2 * 3)
print ('Division: ', 4 / 2)                    # Division in
python gives floating number
print('Division: ', 6 / 2)
print('Division: ', 7 / 2)
```

```
print('Division without the remainder: ', 7 // 2)   # gives without
the floating number or without the remaining
print('Modulus: ', 3 % 2)                           # Gives the
remainder
print ('Division without the remainder: ',7 // 3)
print('Exponential: ', 3 ** 2)                      # it means 3 * 3
```

```
Addition:  3
Subtraction:  1
Multiplication:  6
Division:  2.0
Division:  3.0
Division:  3.5
Division without the remainder:  3
Modulus:  1
Division without the remainder:  2
Exponential:  9
```

```
x = 16
y = 3

print('x + y =',x+y) # 19
print('x - y =',x-y) # 13
print('x * y =',x*y) # 48
print('x / y =',x/y) # 5.333
print('x // y =',x//y) # 519
```

```
x + y = 19
x - y = 13
x * y = 48
x / y = 5.333333333333333
x // y = 5
```

```
# Python natively allows (nearly) infinite length integers while
floating point numbers are double precision numbers:
22**600
```

```
284189804538526226038838564947097313112181583622932247004455439407966437769117881295085772463099292580695984678960082875960538630821097468019477222807837200326498181984644523753583486191200211357972518570195570325656386596677454211151944685590215412690274387467884863926495814053275166587183170114538584652187798809721406125515041497010821641793674841985707342300035283558265456442237947799117239171158452137399213278204090054607085479424452349064988805884112767551200021089635109552765346305229659661779516260380639949371812289388442526680985153850457184286635477881221737987047175109793197498340834190043552220120437158551236152496820395878444312462690176127282917107188213888406272774071898635097899293391024066551157868996968299020322826669834268971117502088391528305736885128319846020853605724858613 76
```

# 2. Comparison/Relational operators

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

| Symbol | Task Performed | Meaning | Example |
|---|---|---|---|
| **>** | greater than | True if left operand is greater than the right | **x > y** |
| **<** | less than | True if left operand is less than the right | **x < y** |
| **==** | equal to | True if both operands are equal | **x == y** |
| **!=** | not equal to | True if both operands are not equal | **x != y** |
| **>=** | greater than or equal to | True if left operand is greater than or equal to the right | **x >= y** |
| **<=** | less than or equal to | True if left operand is less than or equal to the right | **x <= y** |

Note the difference between **==** (equality test) and **=** (assignment)

```
print(6 > 3)                                # True, because 3 is greater
than 2
print(6 >= 3)                               # True, because 3 is greater
than 2
print(6 < 3)                                # False,  because 3 is greater
than 2
print(3 < 6)                                # True, because 2 is less than
3
print(3 <= 6)                               # True, because 2 is less than
3
print(6 == 3)                               # False, because 3 is not equal
to 2
print(6 != 3)                               # True, because 3 is not equal
to 2
print(len("apple") == len("avocado"))  # False
print(len("apple") != len("avocado"))  # True
print(len("apple") < len("avocado"))   # True
```

```
print(len("banana") != len("orange"))  # False
print(len("banana") == len("orange"))  # True
print(len("tomato") == len("potato"))  # True
print(len("python") > len("coding"))   # False
```

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

```
x = 30
y = 22

print('x > y is',x>y)   # False
print('x < y is',x<y)   # True
print('x >= y is',x>=y) # False
print('x <= y is',x<=y) # True
```

```
x > y is True
x < y is False
x >= y is True
x <= y is False
```

```
z = 3  # 3 is assign to variable z
z == 3 # 3 is equal to z
```

```
True
```

## 3. Logical / Boolean Operators

Logical operators are the `and`, `or`, `not` operators.

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

```
print('True == True: ', True == True)
print('True == False: ', True == False)
print('False == False:', False == False)
print('True and True: ', True and True)
print('True or False:', True or False)
```

```
True == True:  True
True == False:  False
False == False: True
True and True:  True
True or False: True
```

```
# Another way comparison

print('1 is 1', 1 is 1)                   # True  - because the data
values are the same
print('1 is not 2', 1 is not 2)           # True  - because 1 is not 2
print('A in Milaan', 'A' in 'Milaan')     # True  - A found in the
string
print('B in Milaan', 'B' in 'Milaan')     # False - there is no
uppercase B
print('python' in 'python is fun')        # True  - because coding for
all has the word coding
print('a in an:', 'a' in 'an')            # True
print('27 is 3 ** 3:', 27 is 3**3)        # True
```

```
1 is 1 True
1 is not 2 True
A in Milaan False
B in Milaan False
True
a in an: True
27 is 3 ** 3: True
```

```
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:9: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:9: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-49-7c9145eb11e9>:3: SyntaxWarning: "is" with a literal.
Did you mean "=="?
  print('1 is 1', 1 is 1)                   # True  - because the data
values are the same
<ipython-input-49-7c9145eb11e9>:4: SyntaxWarning: "is not" with a
literal. Did you mean "!="?
  print('1 is not 2', 1 is not 2)          # True  - because 1 is not
2
<ipython-input-49-7c9145eb11e9>:9: SyntaxWarning: "is" with a literal.
```

```
Did you mean "=="?
  print('27 is 3 ** 3:', 27 is 3**3)       # True

print(6 > 3 and 5 > 3) # True  - because both statements are true
print(6 > 3 and 5 < 3) # False - because the second statement is false
print(6 < 3 and 5 < 3) # False - because both statements are false
print(6 > 3 or 5 > 3)  # True  - because both statements are true
print(6 > 3 or 5 < 3)  # True  - because one of the statement is true
print(6 < 3 or 5 < 3)  # False - because both statements are false
print(not 6 > 3)       # False - because 6 > 3 is true, then not True
gives False
print(not True)        # False - Negation, the not operator turns true
to false
print(not False)       # True
print(not not True)    # True
print(not not False)   # False

True
False
False
True
True
False
False
False
True
True
False

x = True
y = False

print('x and y is',x and y) # False
print('x or y is',x or y) # True
print('not x is',not x) # False

x and y is False
x or y is True
not x is False

True and (not(not False)) or (True and (not True))  # What will be
output?

False
```

## 4. Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates **bit by bit**, hence the name. **For example:** 2 is `10` in binary and 7 is `111`. **In the table below:** Let `x` = 10 (`0000 1010` in binary) and `y` = 4 (`0000 0100` in binary)

| Operator | Meaning | Symbol | Task Performed | Example |
|---|---|---|---|---|
| **and** | Logical and | **&** | Bitwise And | **x & y** = 0 (**0000 0000**) |
| **or** | Logical or | **\|** | Bitwise OR | **x \| y** = 14 (**0000 1110**) |
| **not** | Not | **~** | Bitwise NOT | **~x** = -11 (**1111 0101**) |
| | | **^** | Bitwise XOR | **x ^ y** = 14 (**0000 1110**) |
| | | **>>** | Bitwise right shift | **x >> 2** = 2 (**0000 0010**) |
| | | **<<** | Bitwise left shift | **x << 2** = 40 (**0010 1000**) |

```
a = 2 #binary: 0010
b = 3 #binary: 0011
print('a & b =',a & b,"=",bin(a&b))

a & b = 2 = 0b10

5 >> 1

# 0 ⇨ 0000 0101
#       0000 0010
# 0010 is 2 in decimal

# Explanation
# 0000 0101 -> 5 (5 is 0101 in binary)
# Shifting the digits by 1 to the right and zero padding that will be:
# 0 ⇨ 0000 0101 = 0000 0010
# 0000 0010 -> 2

2
```

## 5. Assignment Operators

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

| Symbol | Example | Equivalent to |
|---|---|---|
| **=** | **x = 5** | **x = 5** |
| **+=** | **x += 5** | **x = x + 5** |
| **-=** | **x -= 5** | **x = x - 5** |
| ***=** | **x *= 5** | **x = x * 5** |
| **/=** | **x /= 5** | **x = x / 5** |
| **%=** | **x %= 5** | **x = x % 5** |
| **//=** | **x //= 5** | **x = x // 5** |

| Symbol | Example | Equivalent to |
|---|---|---|
| **= | **x \|= 5** | **x = x \| 5** |
| **&=** | **x &= 5** | **x = x & 5** |
| \|= | **x \| | = 5** |
| **^=** | **x ^= 5** | **x = x ^ 5** |
| **>>=** | **x >>= 5** | **x = x >> 5** |
| **<<=** | **x <<= 5** | **x = x << 5** |

The binary operators can be combined with assignment to modify a variable value.

```
x = 1
x += 2 # add 2 to x
print("x is",x)
x <<= 2 # left shift by 2 (equivalent to x *= 4)
print('x is',x)
x **= 2 # x := x^2
print('x is',x)

x is 3
x is 12
x is 144
```

# 6. Special Operators

There are some special types of operators like the identity operator or the membership operator.

## 1. Identity Operators

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

| Symbol | Meaning | Example |
|---|---|---|
| **is** | True if the operands are identical (refer to the same object) | **x is True** |
| **is not** | True if the operands are not identical (do not refer to the same object) | **x is not True** |

```
x1 = 6
y1 = 6
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3] # list
y3 = [1,2,3] # list
```

```
# Output: False
print(x1 is not y1)

# Output: True
print(x2 is y2)

# Output: False because two list [] can never be equal
print(x3 is y3)
```

```
False
True
False
```

## 2. Membership Operators

**`in`** and **`not in`** are the membership operators in Python. They are used to test whether a value or variable is found in a **sequence** (string, list, tuple, and dictionary).

| Symbol | Meaning | Example |
|---|---|---|
| **`in`** | True if value/variable is found in sequence | **5 in x** |
| **`not in`** | True if value/variable is not found in sequence | **5 not in x** |

```
x = 'Hello world'
y = {1:'a',2:'b'} # dictionary 1 is key and 'a' is element. So we
access element without its key.

# Output: True
print('H' in x)  # Do we have 'H' in 'Hello World' ?

# Output: True
print('hello' not in x)  # Do we have 'hello' in 'Hello World' ?

# Output: True
print(1 in y)

# Output: False because we cannot identify 'a' without its key hence
it is Flase.
print('a' in y)
```

```
True
True
True
False
```

# Control Flow Statements

Decision-making statements are those that help in deciding the **flow of the program**.

For example, at times, you might want to execute a block of code only if a particular condition is satisfied. Well, in this case, decision-making statement will be of great help.

The flow control statements are divided into three categories:

1. Conditional statements
2. Iterative statements
3. Transfer/Control statements

## `if` Statement

The `if` statement is the simplest form. It takes a condition and evaluates to either `True` or `False`.

```
if(condition):
    statement 1
    statement 2
    statement n
```

```
# Example 1:
grade = 70
if grade >= 65:
    print("Passing grade")
```

```
Passing grade
```

```
# Example 2: If the number is positive, we print an appropriate
message

num = 3
if (num > 0):  # if condition is TRUE: enter the body of if
    print(num, "is a positive number.")
print("This is always printed.")

num = -1
if num > 0:  # if condition is FALSE: do not enter the body of if
    print(num, "is a negative number.")
print("This is also always printed.")
```

```
3 is a positive number.
This is always printed.
This is also always printed.
```

```
number = 9
if number > 6:
    # Calculate square
```

```
    print(number * number)
print('Next lines of code')
```

```
81
Next lines of code
```

```
# Example 4: Check if num1 is less than num2

num1, num2 = 6, 9
if(num1 < num2):
    print("num1 is less than num2")
```

```
num1 is less than num2
```

Well, I also want to give you a nice shortcut!

## Shortcut for `if` statement (Short-hand `if` or one-line `if`)

If you have only one statement to execute, then you can put it on the same line as the if statement.

```
num1, num2 = 5, 6
if(num1 < num2): print("num1 is less than num2")
```

```
num1 is less than num2
```

## `if-else` Statement

It is likely that we will want the program to do something even when an if statement **evaluates to false**.

```
if condition:
    statement 1
else:
    statement 2

# Example 1:

grade = 60

if grade >= 65:
    print("Passing grade")
else:
    print("Failing grade")
```

```
Failing grade
```

```
# Example 2: Program checks if the number is positive or negative and
displays an appropriate message
```

```
num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```

```
Positive or Zero
```

```
# Example 3: program to check if a num1 is less than num2

num1, num2 = 6, 5
if (num1 < num2):
    print("num1 is less than num2")
else:
    print("num2 is less than num1")
```

```
num2 is less than num1
```

```
# Example 4:

def password_check(password):
    if password == "Python@99>":
        print("Correct password")
    else:
        print("Incorrect Password")

password_check("Python@99>")
# Output Correct password

password_check("Python99")
# Output Incorrect Password
```

```
Correct password
Incorrect Password
```

I'll give a shortcut for this too! (Thank me later)

## Shortcut of `if else` (Short Hand `if ... else` or One line `if else`)

If you have **only one statement each for if and else, then they can be put in the same line**. This can be done as shown below

```
hungry = True
x = 'Feed the bear now!' if hungry else 'Do not feed the bear.'
print(x)
```

```
Feed the bear now!
```

```
a = 3
print('A is positive') if a > 0 else print('A is negative') # first
condition met, 'A is positive' will be printed
```

```
A is positive
```

```
num1, num2 = 6, 5
print("num1 is less than num2") if (num1 < num2) else print("num2 is
less than num1")
```

```
num2 is less than num1
```

```
number = 96
if number > 0: print("positive")
else: print("negative")
```

```
positive
```

## `if-elif-else` Statement

`elif` keyword is used to chain multiple conditions one after another.

```
if condition-1:
     statement 1
elif condition-2:
     stetement 2
elif condition-3:
     stetement 3
     ...
else:
     statement
```

```
# Example 1:
'''In this program, we check if the number is positive or negative or
zero and
display an appropriate message'''

num = 0

# Try these two variations as well:
# num = 0
# num = -4.5

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
```

```
else:
    print("Negative number")
```

```
Zero
```

```
# Example 2:

num1, num2 = 5, 5
if(num1 > num2):
    print("num1 is greater than num2")
elif(num1 == num2):
    print("num1 is equal to num2")
else:
    print("num1 is less than num2")
```

```
num1 is equal to num2
```

```
# Example 3:

x = 10
y = 12
if x > y:
    print("x>y")
elif x < y:
    print("x<y")
else:
    print("x=y")
```

```
x<y
```

```
# Example 4:

grade = 96

if grade >= 90:
    print("A grade")
elif grade >=80:
    print("B grade")
elif grade >=70:
    print("C grade")
elif grade >= 65:
    print("D grade")
else:
    print("Failing grade")
```

```
A grade
```

## Nested `if` Statement

We can have a nested-if-else or nested-if-elif-else statement inside another `if-else` statement. This is called nesting in computer programming. The nested if statements is useful when we want to make a series of decisions.

```
if conditon_outer:
    if condition_inner:
        statement of nested if
    else:
        statement of nested if else:
    statement ot outer if
else:
    Outer else
```

```
# Example 1:

a=10
if a>=20:  # Condition FALSE
    print ("Condition is True")
else:  # Code will go to ELSE body
    if a>=15:  # Condition FALSE
        print ("Checking second value")
    else:  # Code will go to ELSE body
        print ("All Conditions are false")
```

```
All Conditions are false
```

```
# Example 2:

x = 10
y = 12
if x > y:
    print( "x>y")
elif x < y:
    print( "x<y")
    if x==10:
        print ("x=10")
    else:
        print ("invalid")
else:
    print ("x=y")
```

```
x<y
x=10
```

## `for` Loop

Python loops are used to repeatedly execute a block of statements until a given condition returns to be `False`.

```
for element in sequence:
    body of for loop

# Example 1: For loop
words = ['one', 'two', 'three', 'four', 'five']

for i in words:
    print(i)
```

```
one
two
three
four
five
```

```
# Example 2: Calculate the average of list of numbers
numbers = [10, 20, 30, 40, 50]

# definite iteration
# run loop 5 times because list contains 5 items
sum = 0
for i in numbers:
    sum = sum + i
list_size = len(numbers)
average = sum / list_size
print(average)
```

```
30.0
```

## `while` Loop

The `while` loop in Python is used to iterate over a block of code as long as the expression/condition is `True`.

```
while condition:
    body of while loop

# Example 1: Print numbers less than 5
count = 1
# run loop till count is less than 5
while count < 5:
    print(count)
    count = count + 1
```

```
1
2
3
4
```

```
# Example 2:
num = 10
```

```
sum = 0
i = 1
while i <= num:
    sum = sum + i
    i = i + 1
print("Sum of first 10 number is:", sum)
```

```
Sum of first 10 number is: 55
```

## Control Statements

As you may know, loops in Python are used to iterate repeatedly over a block of code. But at times, you might want to **shift the control once a particular condition is satisfied**.

| | Statement | Description |
|---|---|---|
| 1 | **`break`** | Terminate the current loop. Use the **`break`** statement to come out of the loop instantly. |
| 2 | **`continue`** | Skip the current iteration of a loop and move to the next iteration |
| 3 | **`pass`** | Do nothing. Ignore the condition in which it occurred and proceed to run the program as usual |

Loops iterate over a block of code until the test expression is `False`, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

```
# Example: break statement

numbers = [10, 30, 120, 330]
for i in numbers:
    if i > 100:
        break
    print('current number', i)
```

```
current number 10
current number 30
```

```
# Example: continue statement

for val in "string":
    if val == "i":
        continue
    print(val)
```

```
print("The end")
```

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

```
# Example: pass statemetn
months = ['January', 'June', 'March', 'April']
for mon in months:
    pass
print(months)
```

```
['January', 'June', 'March', 'April']
```

# Functions

a **function is a block of organized, reusable (DRY- Don't Repeat Yourself) code with a name** that is used to perform a single, specific task. It can take arguments and returns the value.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it improves efficiency and reduces errors because of the reusability of a code.

```
def function_name(parameter1, parameter2):
    """docstring"""
    # function body
    # write some action
  return value

# Example 1:
def greet():
    print("Welcome to Python for Data Science")

# call function using its name
greet()
```

```
Welcome to Python for Data Science
```

```
# Example 2:
def add_two_numbers ():
    num_one = 3
    num_two = 6
    total = num_one + num_two
    print(total)
```

```
add_two_numbers() # calling a function
```

```
9
```

```
def course(name, course_name):
    print("Hello", name, "Welcome to Python for Robotics")
    print("Your course name is", course_name)

course('Arthur', 'Python')   # call function
```

```
Hello Arthur Welcome to Python for Robotics
Your course name is Python
```

# OOPs Concepts

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

## Class

In Python, everything is an object. A class is a blueprint for the object. To create an object we require a model or plan or blueprint which is nothing but class.

```
class classname:
    '''documentation string'''
    class_suite

# Creating a class

class Person:
    pass
print(Person)
```

```
<class '__main__.Person'>
```

## Object

The physical existence of a class is nothing but an object. In other words, the object is an entity that has a state and behavior.

`reference_variable = classname()`

```
# Example 1: We can create an object by calling the class
```

```
p = Person()
print(p)
```

```
<__main__.Person object at 0x78899aa35600>
```

```
# Example 2: Creating Class and Object in Python

class Student:
    """This is student class with data"""
    def learn(self):    # A sample method
        print("Welcome to My. Keivalya Pandya's class on Python
Programming")

stud = Student()        # creating object
stud.learn()            # Calling method
```

```
Welcome to My. Keivalya Pandya's class on Python Programming
```

## Class Constructor

In the examples above, we have created an object from the `Person` class. However, a class without a constructor is not really useful in real applications. Let us use constructor function to make our class more useful.

```
class Person:
      def __init__ (self, name):
        # self allows to attach parameter to the class
          self.name = name

p = Person('Kv')
print(p.name)
print(p)
```

```
Kv
<__main__.Person object at 0x78899aa35630>
```

Let's add more parameters, just for fun

```
# Example 1: add more parameters to the constructor function.

class Person:
      def __init__(self, firstname, lastname, age, country, city):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city

p = Person('Keivalya', 'Pandya', 22, 'India', 'India')
```

```
print(p.firstname)
print(p.lastname)
print(p.age)
print(p.country)
print(p.city)
```

```
Keivalya
Pandya
22
India
India
```