1. Basic Concepts:--------->

```
<img src="01-images/Basics-01-overview.png">
<img src="01-images/Basics-02-overview.png">
```

In [ ]:
```

```

In [ ]:
```

```

1.1. My first Program:

Let's start off by creating a short program that displays "Hello World!". In Python, we use the **print** statement to output text:

```
first explain in idle python
then install with pip
then tell about jupyter notebook
then explain in jupyter notebook
```

In [3]:
```
print("Hello World!")
```

```
Hello World!
```

In [4]:
```
print("My name is Nikky Amresh.\nI love to code.")
```

```
My name is Nikky Amresh.
I love to code.
```

In [5]:
```
print(101)
```

```
101
```

In [ ]:
```

```

1.2. Datatypes and Variables:

In [4]:
```
a = 10          #integer
print(type(a))
```

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

In [5]:
```
b = 10.5        #float
print(type(b))
```

```
<class 'float'>
```

In [6]:
```
c = "Hello World!"   # string
print(type(c))
```

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

In python, the datatypes are mainly organised into five types:

1. None Type: An object that does not contain any value.
2. Numeric Types: Integer, Float, complex
3. Sequences: string, bytes, bytearray, list, tuple, range
4. Sets: ex:-> sets
5. Mapping: ex:-> dictionary
6. Boolean Types: True and False

1.2.1. None:

The **None** object is used to represent the absence of a value. It is similar to null in other programming languages. Like other "empty" values, such as 0, [] and the empty string, it is False when converted to a Boolean variable. When entered at the Python console, it is displayed as the empty string.

In [7]:
```
#Implementation of datatypes:
a = None   # None type
print(a)
print(type(a))
```

```
None
<class 'NoneType'>
```

In [1]:
```
a = bool(None)
print(a)
```

```
False
```

1.2.2. Numeric Types:

In [8]:
```
b = 20          # integer
print(type(b))
```

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

In [9]:
```
c = -20          # integer
print(type(c))
```

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

In [10]:
```
d = 20.52        # float
print(type(d))
```

```
<class 'float'>
```

In [11]:
```
e = 210.0        # float
print(type(e))
```

```
<class 'float'>
```

In [12]:
```
f = 2 + 3j       # complex
print(f)
print(type(f))
```

```
(2+3j)
<class 'complex'>
```

In [13]:
```
g = 0.8j     # complex
print(g)
type(g)
```

```
0.8j
```

Out[13]:
```
complex
```

1.2.3. Sequences:

1.2.3.1. String:

If you want to use text in Python, you have to use a string. A string is created by entering text between two single or double quotation marks.

When the Python console displays a string, it generally uses single quotes. The delimiter used for a string doesn't affect how it behaves in any way.

In [14]:
```
h = 'Python is fun!'     # string
print(type(h))
```

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

In [15]:
```
i = "My name is jacob.\nI love to code"     # string
print(i)
type(i)
```

```
My name is jacob.
I love to code
```

Out[15]:
```
str
```

In [6]:
```
j = """Hello Raj....
How are you?"""          # docstring
print(j)
type(j)
```

```
Hello Raj....
How are you?
```

Out[6]:
```
str
```

```
More about string we will study in upcoming modules.
```

In [ ]:
```

```

1.2.3.2. List:

Lists are another type of object in Python. They are used to store an indexed list of items. A list is created using **square brackets** with **commas** separating items. The certain item in the list can be accessed by using its index in square brackets.

In [17]:
```
words = ["Hello", "Jacob", "Good"]
print(words)
```

```
['Hello', 'Jacob', 'Good']
```

In [18]:
```
type(words)
```

Out[18]:
```
list
```

In [19]:
```
print(words[0])
print(words[1])
print(words[2])
```

```
Hello
Jacob
Good
```

In [20]:
```
empty_list = []
print(empty_list)
```

```
[]
```

Typically, a list will contain items of a single item type, but it is also possible to include several different types.

Lists can also be nested within other lists.

In [21]:
```
number = 3
things = ["string", 0, [1, 2, number], 4.56]
print(things[1])
print(things[2])
print(things[2][2])
```

```
0
[1, 2, 3]
3
```

Lists of lists are often used to represent 2D grids, as Python lacks the multidimensional arrays that would be used for this in other languages.

Indexing out of the bounds of possible list values causes an IndexError.

Some types, such as strings, can be indexed like lists. Indexing strings behaves as though you are indexing a list containing each character in the string.

For other types, such as integers, indexing them isn't possible, and it causes a TypeError.

In [22]:
```
s = "Hello world!"
print(s[6])
```

```
w
```

In [23]:
```
print(s[20])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-23-96e510af8929> in <module>
----> 1 print(s[20])

IndexError: string index out of range
```

```
More about lists we will study in upcoming modules.
```

In [ ]:
```

```

1.2.3.3. Tuple:

Tuples are very similar to lists, except that they are immutable (they cannot be changed).

Also, they are created using parentheses, rather than square brackets.

In [24]:
```
words = ("spam", "eggs", "sausages",)
print(words)
```

```
('spam', 'eggs', 'sausages')
```

- You can access the values in the tuple with their index, just as you did with lists:

In [25]:
```
print(words[0])
```

```
spam
```

- Trying to reassign a value in a tuple causes a TypeError.

In [26]:
```
words[1] = "cheese"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-bf923172c036> in <module>
----> 1 words[1] = "cheese"

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

- Tuples can be created without the parentheses, by just separating the values with commas.
- Tuples are faster than lists, but they cannot be changed

In [27]:
```
my_tuple = "one", "two", "three"
print(my_tuple[0])
```

```
one
```

- An empty tuple is created using an empty parenthesis pair.

In [28]:
```
t = ()
type(t)
```

Out[28]:
```
tuple
```

In [29]:
```
t1 = (2)
type(t1)
```

Out[29]:
```
int
```

In [30]:
```
t2 = (2,)
type(t2)
```

Out[30]: tuple

In [ ]:
```

```

```
##### 1.2.3.4. range:

<img src = "01-images/syntax-range.png">
```

In [31]:
```
n = range(10)
print(n)
```

```
range(0, 10)
```

In [32]:
```
n = list(range(10))
print(n)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- The call to list is necessary because range by itself creates a range object, and this must be converted to a list if you want to use it as one.

In [33]:
```
n = list(range(10, 20))
print(n)
```

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

- range can have a third argument, which determines the interval of the sequence produced. This third argument must be an integer.

In [34]:
```
n = list(range(10, 31, 2))
print(n)
```

```
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

In [35]:
```
n = list(range(0,20.2,2))    # we can't take float value inside range functi
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-3d21a4fac1d9> in <module>
----> 1 n = list(range(0,20.2,2))    # we can't take float value inside range f
unction

TypeError: 'float' object cannot be interpreted as an integer
```

In [ ]:
```
1
```

1.2.4. Sets:

- Sets are data structures, similar to lists or dictionaries.
- They are created using curly braces, or the set function.
- They share some functionality with lists, such as the use of in to check whether they contain a particular item.
- To create an empty set, you must use set(), as {} creates an empty dictionary.

In [36]:
```
1 s = {1, 2, 3, 1, 4, 5, 2, 6}
2 print(s)
```

```
{1, 2, 3, 4, 5, 6}
```

- Sets differ from lists in several ways, but share several list operations such as len.
- They are unordered, which means that they can't be indexed.
- They cannot contain duplicate elements.
- Due to the way they're stored, it's faster to check whether an item is part of a set, rather than part of a list.

```
1 sets have many functions.....show them using tab. Further we will study in
  upcoming modules.
```

In [ ]:
```
1
```

In [ ]:
```
1
```

1.2.5. Mapping: (Dictionaries)

- Dictionaries are data structures used to map arbitrary keys to values.
- Lists can be thought of as dictionaries with integer keys within a certain range.
- Dictionaries can be indexed in the same way as lists, using square brackets containing keys.
- Each element in a dictionary is represented by a key:value pair.

In [37]:
```
1 ages = {"Dave": 24, "Mary": 42, "John": 58}
2 print(ages["Dave"])
3 print(ages["Mary"])
```

```
24
42
```

In [38]:
```
d = {1: 24, 2: 25, 3: 102}
print(d)
```

```
{1: 24, 2: 25, 3: 102}
```

- An empty dictionary is defined as {}

In [39]:
```
empty_dict = {}
print(empty_dict)
```

```
{}
```

1.2.6. Boolean Types:

- Another type in Python is the Boolean type. There are two Boolean values: True and False.
- They can be created by comparing values, for instance by using the equal operator ==.

In [40]:
```
my_boolean = True
print(my_boolean)
type(my_boolean)
```

```
True
```

Out[40]:
```
bool
```

In [41]:
```
2 == 3
```

Out[41]:
```
False
```

In [42]:
```
2 != 3
```

Out[42]:
```
True
```

- Be careful not to confuse assignment (one equals sign) with comparison (two equals signs).

In [43]:
```
print(7 > 5)          # comparikson
```

```
True
```

In [17]:
```
b = bool()
print(b)          # empty value hence false
```

```
False
```

In [18]:
```
b = bool(10)
print(b)
```

```
True
```

In [19]:
```
b = bool([])
print(b)
```

```
False
```

In [21]:
```
b = bool([0])
print(b)
```

```
True
```

In [23]:
```
b = bool(0)
print(b)
```

```
False
```

In [ ]:
```

```

- We can also create these sequences by the name of the sequence functions. For example:

```
list        ------- >   list()
string      ------- >   str()
tuple       ------- >   tuple()
set         ------- >   set()
dictionary  ------- >   dict()
range       ------- >   range()
```

  we can also use these as type conversion functions. we will study these in next section.

In [11]:
```
lst = list()
print(lst)
```

```
[]
```

In [12]:
```
string = str()
print(string)
```

In [14]:
```
tup = tuple()
print(tup)
```

```
()
```

In [15]:
```
s = set()
print(s)
```

```
set()
```

In [16]:
```
d = dict()
print(d)
```

```
{}
```

variables:

- Variables names must start with a letter or an underscore, such as: *underscore. underscore*
- The remainder of your variable name may consist of letters, numbers and underscores. password1. n00b. ...
- Names are case sensitive. case_sensitive, CASE_SENSITIVE, and Case_Sensitive are each a different variable.

- correct examples of variables:---> a, b_, _name, na_2_rk_, etc
- Incorrect examples of variables:---> 2a, 21_5, gh@4, etc

In [44]:
```
2a = 5
```

```
  File "<ipython-input-44-1fd2f3417088>", line 1
    2a = 5
     ^
SyntaxError: invalid syntax
```

In [ ]:
```

```

Type conversion functions:

In [45]:
```
# string to integer
a = "5"
print(type(a))
a = int(a)
print(a)
print(type(a))
```

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

In [46]:
```
# float to integer
b = 2.5     #float
b = int(b)
print(b)
print(type(b))
```

```
2
<class 'int'>
```

In [47]:
```
# float to string
a = 20.2
a = str(a)
print(a)
type(a), a
```

```
20.2
```

Out[47]:
```
(str, '20.2')
```

In [48]:
```
# list to tuple
l = [1, 2, 3, 4, 5]
print(l)
print(type(l))
l = tuple(l)
print(l)
print(type(l))
```

```
[1, 2, 3, 4, 5]
<class 'list'>
(1, 2, 3, 4, 5)
<class 'tuple'>
```

In [ ]:
```

```

1.3. Operators and Operands:

1. Arithematic Operators
2. Assignment operators
3. Comparison operators
4. Logical operators
5. Operator precedence

1.3.1. Arithematic operators:

In [49]:
```
a, b = 10, 6
print("Addition: ", a+b)
print("Subtraction: ", a-b)
print("Mul: ", a*b)
print("Div: ", a/b)
print("Mod: ", a%b)            # remainder
print("Power: ", a**b)
print("Floor Div: ", a//b)     # integer division
```

```
Addition:  16
Subtraction:  4
Mul:  60
Div:  1.6666666666666667
Mod:  4
Power:  1000000
Floor Div:  1
```

In [ ]:
```

```

In [ ]:
```

```

1.3.2. Assignment Operators:

In [50]:
```
a = 10   #(explain it)
```

In [51]:
```
a = 20
print(a)
```

```
20
```

In [52]:
```
a = b = c = 10
print(a, b, c)
```

```
10 10 10
```

In [53]:
```
x, y = 10, 5
print(x, y)
```

```
10 5
```

In [54]:
```
x = 6
print("Before compound assignment operations:", x)
x += 2          # x= 8
print(x)
x *= 4          # x = 32
print(x)
x /= 4          # x = 8.0
print(x)
x -= 4          # x = 4.0
print(x)
x **= 2         # x = 16.0
print(x)

# similarly others
```

```
Before compound assignment operations: 6
8
32
8.0
4.0
16.0
```

In [ ]:
```
1
```

1.3.3. Comparison operators:

In [55]:
```
x, y = 45, 85
print(x == y)
print(x != y)
print(x > y)
print(x >= y)
print(x < y)
print(x <= y)
```

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

In [ ]:
```
1
```

1.3.4. Logical operator:

- Boolean logic is used to make more complicated conditions for if statements that rely on more than one condition.
- Python's Boolean operators are and, or, and not.
- The and operator takes two arguments, and evaluates as True if, and only if, both of its arguments are True. Otherwise, it evaluates to False.

- The or operator also takes two arguments. It evaluates to True if either (or both) of its arguments are True, and False if both arguments are False.
- Unlike other operators we've seen so far, not only takes one argument, and inverts it. The result of not True is False, and not False goes to True.
- Python uses words for its Boolean operators, whereas most other languages use symbols such as &&, || and !.

In [ ]:
```
1
```

```
1 first explain them
```

In [56]:
```
1 x = 20
2 y = 30
3 print(x == 20 and y == 30)
4 print(x == 20 and y == 35)
5 print(x == 20 or y == 35)
6 print(not(x == 20 and y == 30))
```

```
True
False
True
False
```

In [ ]:
```
1
```

1.3.5. Operator precedence:

- Operator precedence is a very important concept in programming. It is an extension of the mathematical idea of order of operations (multiplication being performed before addition, etc.) to include other operators, such as those in Boolean logic.
- The below code shows that == has a higher precedence than or:

In [57]:
```
1 False == False or True
```

Out[57]:
```
True
```

In [58]:
```
1 False == (False or True)
```

Out[58]:
```
False
```

In [59]:
```
1 (False == False) or True
```

Out[59]:
```
True
```

- The following table lists all of Python's operators, from highest precedence to lowest.
- Operators in the same box have the same precedence.

Operator	Description
**	Exponentiation (raise to the power)
~, +, -	Complement, unary plus and minus (method names for the last two are +@ and -@)
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR'
|	Bitwise 'OR'
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparison operators, equality operators, membership and identity operators
not	Boolean 'NOT'
and	Boolean 'AND'
or	Boolean 'OR'
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators

In [ ]:
```
1
```

In [60]:
```
# Question:
# What is the result of this code?
x = 4
y = 2
if not 1 + 1 == y or x == 4 and 7 == 8:
    print("Yes")
elif x > y:
    print("No")
```

```
No
```

In [ ]:
```
1
```

1.4. Input and Output operations:

- To build interactive software application, there is a need of communication between user and application.

- we want the end users to enter some data or to input some data that our application will use and at some point aur application will send some data to the end user or display some output to the user.
- and to perform these input and output operations in a standalone python program we use the **input()** function and the **print()** functions respectively.

1.4.1. print()

- Usually, programs take input and process it to produce output.
- In Python, you can use the print function to produce output. This displays a textual representation of something to the screen.
- print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
- Prints the values to a stream, or to sys.stdout by default.

In [61]:
```
#When a string is printed, the quotes around it are not displayed.
```

In [62]:
```
print("Hello", "World", "I", "Love", "Coding", sep = "***", end = "----->END
```

```
Hello***World***I***Love***Coding----->END
```

In [63]:
```
print("Hello"*3)
```

```
HelloHelloHello
```

- Always new line also added at the end. to skip we can add the end attribute.end=""

In [64]:
```
print("Hello", end = "")
print("World!")
```

```
HelloWorld!
```

print() and string formatting:

In [65]:
```
name = "Raushan"
marks = 33.6
grades = "F"

print("Name is", name, "marks is", marks, "grade is", grades)
```

```
Name is Raushan marks is 33.6 grade is F
```

In [66]:
```
print("Name is %s marks is %0.1f grade is %s"%(name, marks, grades))
```

```
Name is Raushan marks is 33.6 grade is F
```

In [67]:
```
print("Name is {0} Marks is {1} Grades is {2}".format(name, marks, grades))
```

```
Name is Raushan Marks is 33.6 Grades is F
```

In [ ]:
```

```

1.4.2. input():

- To get input from the user in Python, you can use the intuitively named input function.
- The function prompts the user for input, and returns what they enter as a string (with the contents automatically escaped).
- Read a string from standard input. The trailing newline is stripped.

In [68]:
```
a = input("Enter something: ")
print("You entered: ", a)
```

```
Enter something: My name is Nikky
You entered:  My name is Nikky
```

In [69]:
```
a = int(input("Enter a integer: "))
print(a)
type(a)
```

```
Enter a integer: 14
14
```

Out[69]:
```
int
```

In [ ]:
```

```

reading multuple inputs:

In [70]:
```
a, b, c = input("Enter three numbers seperated by space: ").split()
print(a, b, c)
```

```
Enter three numbers seperated by space: 14 15 16
14 15 16
```

In [71]:
```
type(a)
```

Out[71]:
```
str
```

In [72]:
```
a = list(map(int, input("Enter three numbers seperated by space: ").split("
print(a)
```

```
Enter three numbers seperated by space: 14 15 16
[14, 15, 16]
```

In [ ]:
```
1
```