Introduction to Variables and Data Types

Variables

- Are like placeholders to store the data / information and whenever there is a need of using the data / information, we can simply use the variables by calling it.

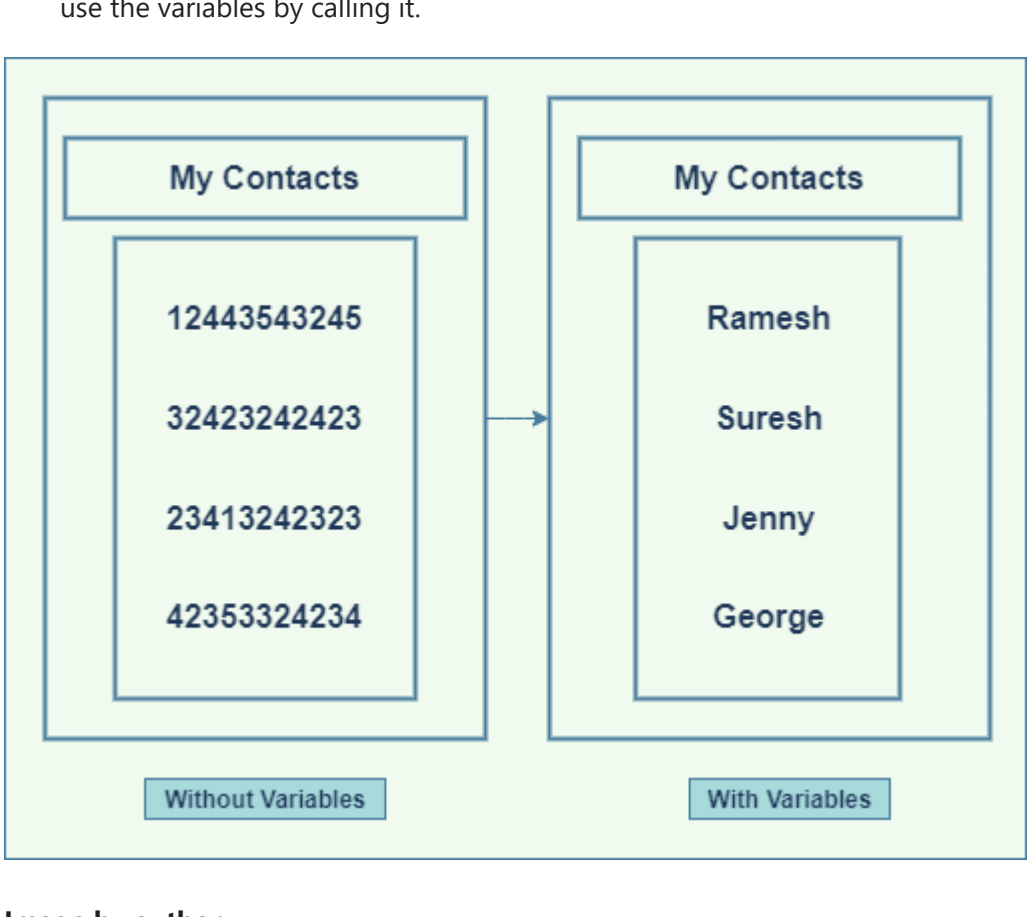

Image by author

Data Types

- Come in handy with variables to tell the compiler or interpreter which ***"type"*** of the ***"data"*** you as a programmer going to store in the variables.

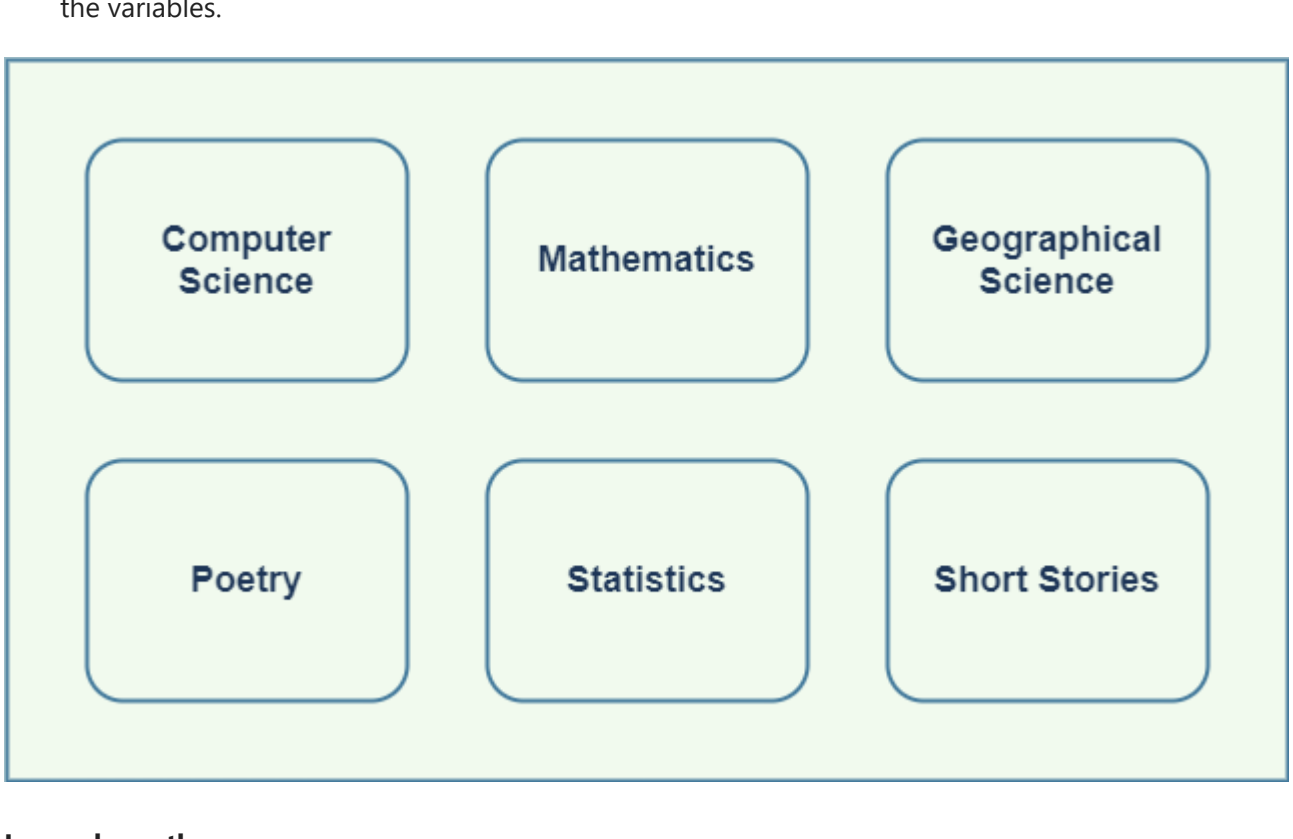

Image by author

Unlike other languages, we need not specify or define the variable like this $\rightarrow$ **(type) [var_name] = (value)**

`C` & `Python` Comparison

C - example

```
int a = 10;
float b = 11.21;
String c = "hi";
```

Python - example (Dynamic type)

```
a = 10
b = 11.21
c = "hi"
d = [1, 2, 3, 4, 5.321, "hello", "world"]
e = True
```

In both the cases, to use the value `10` we can simply call `a`. Same case with `b` and `c`.

`int`

In [1]:

```
a = 123
print(a)
print("a - type", type(a))

b = 4324434355345455
print(b)
print("b - type", type(b))
```

```
123
a - type <class 'int'>
4324434355345455
b - type <class 'int'>
```

`float`

In [2]:

```
a = 12.3423
print(a)
print("a - type", type(a))

b = 21323.000
print(b)
print("b - type", type(b))
```

```
12.3423
a - type <class 'float'>
21323.0
b - type <class 'float'>
```

String → `str`

In [3]:

```
# with single quotes
a = 'sameer'
print(a)
print("a - type", type(a))

# with double quotes
b = "python"
print(b)
print("b - type", type(b))

c = '123'
print(c)
print("c - type", type(c))
```

```
sameer
a - type <class 'str'>
python
b - type <class 'str'>
123
c - type <class 'str'>
```

`list`

- A container that can store elements of **any type** and seperated by comma (,)
  - starts with `[`
  - ends with `]`

In [4]:

```
a = [1, 12.323, "hi"]
print(a)
print("a - type", type(a))
```

```
[1, 12.323, 'hi']
a - type <class 'list'>
```

In [5]:

```
b = [1, 2, 3, ['hey', 12.18]]
print(b)
print("b - type", type(b))
```

```
[1, 2, 3, ['hey', 12.18]]
b - type <class 'list'>
```

Array

- A container that can store elements of **same type** and seperated by comma (,)
  - starts with `[`
  - ends with `]`

In [6]:

```
# int array (list)
a = [1, 2, 3, 4, 5, 6, 7]
print(a)
print("a - type", type(a))

# float array (list)
b = [1.1, 3.324, 4.34, 5.5456, 9.654]
print(b)
print("b - type", type(b))

# string array (list)
c = ["hi", "hola", "hello", "hey"]
print(c)
print("c - type", type(c))

# list inside array_list
d = [[1, 2, 3], [4, 5, 6], [6, 7, 8]]
print(d)
print("d - type", type(d))
```

```
[1, 2, 3, 4, 5, 6, 7]
a - type <class 'list'>
[1.1, 3.324, 4.34, 5.5456, 9.654]
b - type <class 'list'>
['hi', 'hola', 'hello', 'hey']
c - type <class 'list'>
[[1, 2, 3], [4, 5, 6], [6, 7, 8]]
d - type <class 'list'>
```

Note - A list can be an array but an array connot be a list.

List Indexing

- List indexing starts from `0`.
- The last index of the list would be `(n - 1)`.

In [7]:

```
a = ["hey", "python", "sameer", "india", "Earth"]
print(a[0])
print(a[3])
print(a[4])

# the length of the above list is 5
# print(a[5]) # error occurs
```

```
hey
india
Earth
```

`len()`

In [8]:

```
a = ["1", "2", "python", "java", "c", 2121.323]
print(a)
print("a - type", type(a))
######
print("length or size of a is - ", len(a))
```

```
['1', '2', 'python', 'java', 'c', 2121.323]
a - type <class 'list'>
length or size of a is -  6
```

In [9]:

```
len("213121212.213121212")
```

Out[9]: 19

In [10]:

```
len(["213121212.213121212"])
```

Out[10]: 1

`isinstance()` vs `type()`

- `isinstance()` is used to cross check the data type of an object
  - takes two parameters
- `type()` is used to get the actual data type of an object
  - takes one parameters

In [11]:

```
help(isinstance)
```

```
Help on built-in function isinstance in module builtins:

isinstance(obj, class_or_tuple, /)
    Return whether an object is an instance of a class or of a subclass thereof.

    A tuple, as in ``isinstance(x, (A, B, ...))``, may be given as the target to
    check against. This is equivalent to ``isinstance(x, A) or isinstance(x, B)
    or ...`` etc.
```

is "python" an instance of `float`?

In [12]:

```
isinstance("python", float)
```

Out[12]: False

is "hey" an instance of `str`?

In [13]:

```
isinstance("hey", str)
```

Out[13]: True

is [121, 1, 4232] an instance of `list`?

In [14]:

```
isinstance([121, 1, 4232], list)
```

Out[14]: True

Basic Operations

In [15]:

```
v = 12
s = 13
v + s
```

Out[15]: 25

`list` Concatenation

In [16]:

```
my1 = [1, 2, 3, 10, 11, 100, 1000, "boom boom"]
my2 = [4, 5, 6]

print(my1 + my2)
```

```
[1, 2, 3, 10, 11, 100, 1000, 'boom boom', 4, 5, 6]
```

In [17]:

```
my1 = [1, 2, 3, 10, 11, 100, 1000, "boom boom"]
my2 = ["4, 5, 6"]

print(my1 + my2)
```

```
[1, 2, 3, 10, 11, 100, 1000, 'boom boom', '4, 5, 6']
```

In [18]:

```
my1 = [1, 2, 3]
my2 = [4, 5, 6]
print(my1 + my2)
# [5, 7, 9]

# 1) for loops
# 2) numpy methods

my3 = []
for i in range(len(my1)):
    val = my1[i] + my2[i]
    my3.append(val)
print(my3)
```

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

`append()`

- It adds values at the end of the list.

In [19]:

```
my = [1, 2, 3]
# add 4 to my1 in such a way that I get [1, 2, 3, 4]

# 1)
a = [4]
print(my + a)

# 2)
b = 4
# add 4 to my1 in such a way that I get [1, 2, 3, 4]
my.append(b)
print(my)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

In [20]:

```
my3 = [1, 2, 3]
print("before - ", my3)

# append a single value
my3.append(4)
print("single value append - ", my3)

# append an entire list
my3.append(["hi", "hey", "hello"])
print("appending a list - ", my3)

# what is the output of my3 if I print?
print("after - ", my3)
```

```
before -  [1, 2, 3]
single value append -  [1, 2, 3, 4]
appending a list -  [1, 2, 3, 4, ['hi', 'hey', 'hello']]
after -  [1, 2, 3, 4, ['hi', 'hey', 'hello']]
```

`insert()`

- It adds values in the middle of list.

In [21]:

```
my3
```

Out[21]: [1, 2, 3, 4, ['hi', 'hey', 'hello']]

In [22]:

```
some_element = 'wooooo'
```

In [23]:

```
my3.insert(3, some_element)
```

In [24]:

```
print(my3)
print(len(my3))
```

```
[1, 2, 3, 'wooooo', 4, ['hi', 'hey', 'hello']]
6
```

`pop()` and slice

In [25]:

```
my1 = [1, 2, 3, 10, 11, 100, 1000, "mars"]

# pop
print(my1.pop())
print(my1)

print("##########")

# slice
print(my1[1:5])
print(my1)
```

```
mars
[1, 2, 3, 10, 11, 100, 1000]
##########
[2, 3, 10, 11]
[1, 2, 3, 10, 11, 100, 1000]
```

In [26]:

```
s = [1, 2, 3, 4, 5, 6, 7]
s.pop(4)
s
```

Out[26]: [1, 2, 3, 4, 6, 7]

Operators in Python

Represent an action to be performed on the **data / object** and return the result.

Operator is a way to perform an action or operation. Specifically in mathematics if we want to perform addtion operation, we simply use `+`. The concept of Operators is same in Computer science as well.

Types of operators

- Arithematic Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Identity Operators
- Membership Operators
- Bitwise Operators

Arithematic Operators → `+`, `-`, `*`, `/`, `%`, `//`

- `+` → addition
- `-` → subtraction
- `*` → multiplication
- `/` → division
- `%` → modulus
- `//` → integer division or floor division

In [27]:

```
# show example of each kind
a = 123
b = 4
print(a / b)
print(a // b)
```

```
30.75
30
```

In [28]:

```
(12 // 3), (12 / 3)
```

Out[28]: (4, 4.0)

In [29]:

```
e = 4
print(e % 2)
```

```
0
```

In [30]:

```
e / 2
```

Out[30]: 2.0

a^3 → Power Operation

In [31]:

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

```
8
```

BoDMAS

- **B** → ()
- **o** → of
- **D** → /
- **M** → *
- **A** → +
- **S** → -

In [32]:

```
(1 + 21 // 3) * 12 + 13 - 11
```

Out[32]: 98

In [33]:

```
num1 = 32
print(num1 * 3 + 4 - 10)
```

```
90
```

In [34]:

```
12 // (3 * 5)
```

Out[34]: 0

In [35]:

```
289 ** (1/2)
```

Out[35]: 17.0

Assignment Operator → `=`

- Variable declaration (Main use)

In [36]:

```
# show other examples

num2 = 2
num2 = num2 + 4
print("before", num2)

# num2 = num2 + 4
num2 += 4
print("after", num2)
```

```
before 6
after 10
```

In [37]:

```
num2 = 2
print("before", num2)

# num2 = 10 + num2
num2 =+ 10
print("after", num2)
```

```
before 2
after 10
```

Relational Operators

Mainly used in comparing two variables or data values. Always returns `boolean`

```
True
False
```

- `==` → Equal
- `!=` → not Equal
- `<=` → less than or Equal
- `>=` → greater than or Equal
- `<` → less than
- `>` → greater than

In [38]:

```
# show examples → !=

num3 = 1
num4 = -1
num3 != num4
```

Out[38]: True

In [39]:

```
# show examples → ==

num3 = 1
num4 = 1
num3 == num4
```

Out[39]: True

In [40]:

```
# show examples → >

num3 = 1
num4 = 2
num4 > num3
```

Out[40]: True

Logical Operators → `and`, `or`, `not`

To understand this, let us understand a basic chart (sheet) to get started.

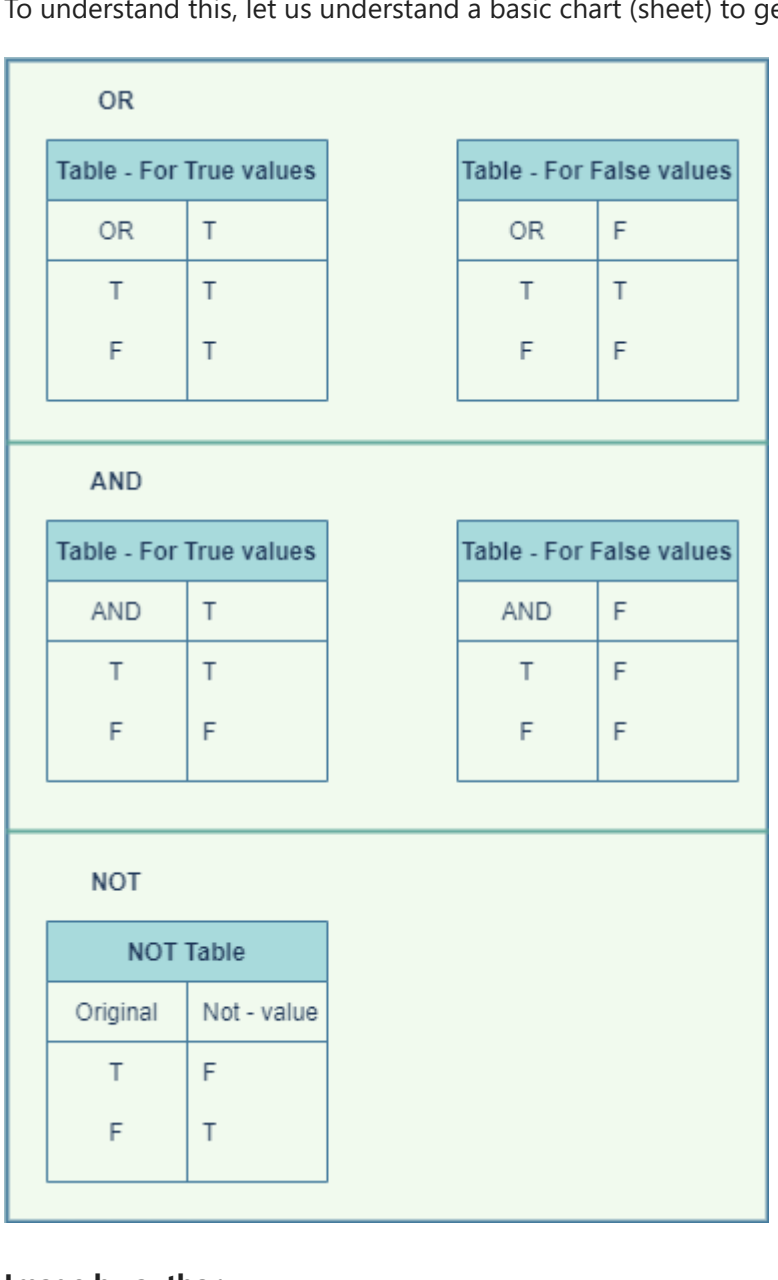

Image by author

Logical operators are heavily used in **conditional statements**. We cannot imagine a development or a program without conditional statements.

Note

- In the above table `T` refers to `True` and `F` refers to `False`.
- For our convenience we can rewrite `T` as `1` and `F` as `0`.

In [41]:

```
True or True
```

Out[41]: True

In [42]:

```
True or False
```

Out[42]: True

In [43]:

```
True or True or False or True
```

Out[43]: True

In [44]:

```
True and False
```

Out[44]: False

In [45]:

```
True and True
```

Out[45]: True

In [46]:

```
True or False and True or False
```

Out[46]: True

In [47]:

```
not True
```

Out[47]: False

In [48]:

```
not False
```

Out[48]: True

In [49]:

```
False or (not (not (not False)))
```

Out[49]: True

Identity Operators → `is`, `is not`

This is used when we want to compare **two objects**. Here I strongly say that we are **not** comparing or relating **two values**. We rather deal with same memory location of the values.

```
>>> 1 is 1
True
>>> 0 is 1
False
```

Note

- **is** → is same as **==**
- **is not** → is same as **!=**

Examples

```
>>> 100 is 100
True
>>> 100 == 100
True
>>> ##########
>>> 100 is 110
False
>>> 110 == 100
False
>>> 110 is not 100
True
>>> 110 != 100
True
```

In [50]:

```
my_num = 100
my_num2 = 110
```

In [51]:

```
print(my_num is my_num2)
print(my_num is not my_num2)
```

```
False
True
```

Membership Operators → `in`, `not in`

It is used to validate if a particular element is present in a sequence or not. It is mostly used `List` to check if an item is present in a sequence. Returns a boolean object after performing the operation.

```
>>> 1 in [1, 2, 3, 4, 5, 6]
True
>>> -1 in [1, 2, -3, -4, -5, 87, 100]
False
```

In [52]:

```
a = ["hi", "hello", "hey"]
"hey" in a
```

Out[52]: True

In [53]:

```
not (not "hola" in a)
```

Out[53]: False

Special Built-In Methods

A built-in method is a special method that can used without its creation. In simple words, it can be used by simply refering its name provided required arguments.

There are so many built-in methods in python which will ease the complications. We need not worry about the code that is been holding the function to work like magic.

List of built-in methods

- `print()`
- `type()`
- `id()`
- `help()`
- `len()`
- `abs()`
- `min()`
- `max()`
- `int()`
- `float()`
- `str()`
- `list()`
- `sorted()`
- `reversed()`
- `round()`
- `...`

Homework or Exercise

Note - Find out more about this by yourself.

- **step 1** - apply `help()` function with any of the above functions.
- **step 2** - read the description of the function.
- **step 3** - come up with your own examples and test it yourself.

You can find one example below -

In [54]:

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In [55]:

```
print("The Earth is my home !!!")
```

```
The Earth is my home !!!
```

What did we learn?

- Variables
- Data types
- Operators
- Types of Opeators
- Built-in methods