Keywords

- Reserved words in Python are known as keywords
- They cannot be used as variable names.
- Their meaning is already reserved.

ex→ True, False, None, and, or, if, else, For, while etc.

Example to ~~see~~ ~~see~~ see all the keywords.

```
import keyword
print(keyword.kwlist)
```

Variable→
A variable is the name give to a memory location in a program.
A variable is a container to store information and retrieve it later.
Nature of store information like numeric, textual, boolean etc.

```
x = 10
```

Variable ↳ assignment operator ↳ information or data

Rules for naming of Variable in Python

① Variable names must start with a letter or an underscore like mysub, _mysub

② The remaining part of variable may consist of letter, underscore and number like mysub1, my_sub1

③ Variable names are case sensitive like my_sub, MY_SUB are different variable.

④ Names cannot begin with a number, Python will throw an error.

⑤ Variable names cannot contain space.

⑥ Variable names cannot contain any symbole.

⑦ Avoid using Python built-in keywords like list, str, def, for etc

⑧ It is consider best practise that names are lowercase with underscores.

Data types

There are following type of data in Python.

1. Numeric
   - 1. integer
     ex- 1, 0, 3, 4
   - 2. Floating point number
     ex- 0.0, 1.0, 3.14
   - 3. complex number
     ex- 3+4j
2. string
3. Boolean
4. None

Python is the dynamic programming language that automatically identifies the type of data.

Integer → Absence of decimal point

ex → ~~1.0~~, 1, 0, 3, 4

Floating point number → presence of decimal points.

ex → 1.0, 0.0, 3.14

complex number → combination of real and imaginary number.

ex → 3+4j

Boolean → it is used to store two values that is True or False

It is used to test whether the result of an expression is True or False.

None → It is used to define a null variable.

a = 34

Identifies a as class <int>

name = "Jagat"

Identifies name as class <string>

c = 27.25

Identifies c as class <float>

d = True or False

Identifies d as class <bool>

e = None

Identifies e as class <None>.

Data structure →

Data structure are fundamental of any programing language, around which program is built.

There are two types of data structure

mutable → Lists, dictionary, sets

↳ To change

Immutable → Numbers, strings, tuples

↳ Not change.

Strings →

String is a sequence of characters enclosed in quotes, single quotes, double quotes or triple quotes.

strings are used in python to record information such as names.

it could either be a word, a phrase, a sentences, a paragraph or an entire cyclopedia.

```
my_sub = "Python"
type(my_sub)
↳ str
print(my_sub)
↳ Python
```

```
phrase = "statistics sits at the heart of m
print(phrase)
type(phrase)
↳ str
```

⇒ find the length of string

```
my_sub = "Machine learning"
len(my_sub)
↳ 16
```

String indexing →

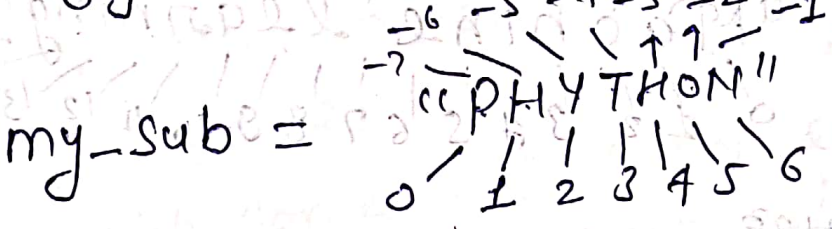
① A string index refer to the location of element present in a string

② we know string are sequence which means python can use indexes to call parts of the sequences.

③ positive indexing start with 0 (zero) and end with length-1 with moving forward

④ Negative indexing start with -1 and end with -length

⑤ In Python, we use brackets [ ] after an object to call its index.




```
my_sub[0]
```

o/p → 'P'

```
my_sub[-3]
```

o/p → 'H'

String slicing

⑥ We can use colon (:) to perform slicing which grabs everything upto a designated point.

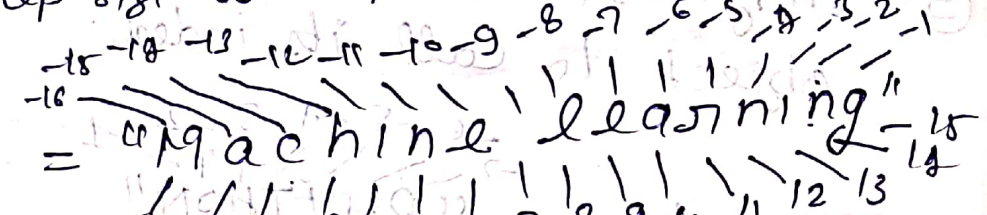
Syntax

```
object [ St : End : Ss]
   or
variable
```

SI → starting index
EI → Ending index
SS → step size

→ starting index is included and ending index excluded.

→ step size is represent direction (forward)



~~my_sub[2:18:3]~~

my_sub[2:13:1]

o/p → chine learn

→ step size is optional

my_sub[0:14]

o/p → maching learni

my_sub[:]

↳ maching learning

→ If you do not specify the ending index, then all elements are extracted which comes after the starting index including at the starting index.

→ The operation knows only to stop when it run through the entire string.

→ If you do not specify the starting index then all elements are extracted which comes before the ending index excluding the element at specified ending index.

→ If you do not specify the starting and ending index. it will extract all the element of the string.

String properties →

① Immutability

↳ This means that once a string is created the element within it can not be changed or not replaced by item assignment.

string = "Hello"

string[0] = "C"

type error

↳ str object does not support item assignment.

② concatenation

↳ addition of two string

string1 = 'abc'

string2 = 'def'

print(string1 + string2)

o/p → 'abcdef'

string function and methods →

algorithm = "Neural Network"

len() function returns the length of string

len(algorithm)

↳ 14

lower()

↳ This methods converts the string to lowercase.

algorithm.lower()

o/p → neural network

upper()

↳ This method converts the string into uppercase.

algorithm.upper()

↳ NEURAL NETWORK.

count()

↳ This method returns the count of a string in the given string.

Syntax

↳ object.count(value, start, end)

algorithm.count('NLe')

o/p → 2

find() →
This method returns the index of the first occurrence of a string present in a given string.

object.find(value, start, end)

algorithm.find('K')

o/p → 13

An important point to note here is that if the string which you are looking for, is not contained in the original string, then the find method will return a value of -1.

replace()
↳ This method takes two arguments.
① The string to replace ② the string to replace with, and returns a modified string after the operation.

algorithm.replace(" ", '_')

~~Neural~~ —
Neural — Network

LIST

① List can be thought of the most general version of a sequence in Python.

② Unlike strings, they are mutable means the item inside a list can be changed.

③ Lists are constructed with brackets [ ] and commas separating every element in the list.

```
my_list = [1, 2, 3, 4, 5]
print(my_list)
o/p→ [1, 2, 3, 4, 5]
type(my_list)
↳ list
```

List can actually hold different object types.

```
my_list = ["Sagar", 1.0, 4, True, None]
type(my_list)
↳ list
```

Indexing

① A list index refer to the location of an element in the list

② positive indexing start from 0 and end with length-1

③ Negative indexing start from -1 and end with -length.

```
                -4   -3   -2   -1
                 ↑    ↑    ↑    ↑
my_list = [ 77,  78,  28,  74]
            0/   1/   2/   3/
```

```
my_list[0]
```

o/p → 77

```
my_list[-2]
```

o/p → 28

list slicing

grab the data upto designated point is known as slicing.

The syntax of slicing

object [SL : EL : SS]

SL → starting index

EL → ending index

SS → step size

Starting index included

ending index excluded

Step size represent step and direction

my_list1 = [100, 200, 50, 90, 300]

my_list1 [0 : 4 : 1]

o/p → [100, 200, 50, 90]

⇒ step size is optional

my_list1 [0 : 4]

↳ [100, 200, 50, 90]

my_list1[:]

o/p → [100, 200, 50, 90, 300]

my_list1[:4]

o/p → [100, 200, 50, 90]

my_list1[0:]

o/p → [100, 200, 50, 90]

⇒ If you do not specify the ending index, then all elements are extracted which comes after the starting index, including the element at that starting index.

⇒ If you do not specify the starting index, then all elements are extracted which comes befores the ending index. excluding the element at the specified ending index

⇒ If you do not specify the starting and ending index. it will extract all the element of the list.

⇒ It should also be noted that list indexing will return an error if there is no element at that index.

List properties

① mutability

↳ So after construction of list we can change it

my_list2 = [100, 79, 49, 50, 97]

~~my_list[~~

my_list2[0] = 500

my_list

o/p → [500, 79, 49, 50, 97]

② Concatenation → addition of list

list 1 = [79, 89, 400]

list2 = [500, 79, 89]

my_list = list1 + list2

print(my_list)

↳ [79, 89, 400, 500, 79, 89]

⇒ List functions

len()

↳ This function return the length of the list

my_list = [7, 8, 9, 10]

len(my_list)

o/p → 4

min()

↳ This function return the minimum element of the list

↳ This function only works with lists of similar data types.

min(my_list)

o/p → 7

max()

↳ This function returns the maximum element of list

↳ This function only works with list of similar data types.

```
x = [78, 21, 24, 99, 33]
max(x)
```

O/P → 99

sum()

↳ This function returns the sum of the element of the list

This function only works with list of numeric data types.

```
my_list = [4, 5, 6, 23]
sum(my_list)
```

O/P → 20

Sorted()

↳ This function returns the sorted list means increasing or decreasing order.

↳ sorted() function takes reverse boolean as an argument.

↳ This function only works on a list with similar data types.

my_list = [4, 7, 8, 10, 11]

sorted(my_list, reverse = True)

o/p → [11, 10, 8, 7, 4]

sorted() function does not change our list.

list methods

append()

① use append() method to permanently add an item to the end of the list

② append() method takes the element which you want to add to the list as an argument.

③ append() method change original list

```
my_list = [4, 5, 6, 2, 3]

my_list.append(9)
```

O/P → [4, 5, 6, 2, 3, 9]

extend()

① use the extend() method to merge a list to an existing list

② extend() method takes a list or any iterable as an argument.

```
my_list = [4, 5, 6, 2, 3]

my_list.extend(['Jagat', 'apple'])

o/p→ [4, 5, 6, 2, 3, 'Jagat', 'apple']
```

pop()

- use pop() to 'pop off' an item from the list.
- By default pop() takes off the element at the last index, but you can also specify which index to pop off.
- pop() takes the index as an argument and returns the element which was popped off.

```
my_list = [1, 2, 3, 4, 5]

my_list.pop()

→ 5
```

remove()

- use remove() to remove an item from the list.
- By default remove() method remove the specified element from the list
- remove() takes the element as an argument.

```
my_list = [1, 2, 3, 4, 5]
my_list.remove(5)
my_list
o/p→ [1, 2, 3, 4]
```

count()

The count() method return the total occurance of a specified element in the list.

```
my_list = [1, 2, 1, 1, 3, 1, 4, 1]
my_list.count(1)
```

→ 5

index()

The index() method returns the index of a specified element.

```
my_list = [4, 5, 6, 7, 8]
my_list.index(4)
  0
```

Syntax

```
list_name.index(element, start, end)
```

sort()

- use sort() to sort the list in either ascending / descending order
- The sorting is done in ascending order by default.
- sort() method takes the reverse boolean as an argument.
- sort() method only works on a list with element of same data type.

```
new_list = [6, 9, 1, 3, 5]
new_list.sort()
```

new_list

↳ o/p → [1, 3, 5, 6, 9]

new_list.sort(reverse = True)

↳ [9, 6, 5, 3, 1]

reverse()

reverse() method reverse the list

my_list = [4, 5, 6, 2, 3]

my_list.reverse()

o/p → [3, 2, 6, 5, 4]

Nested lists

A great feature of Python data structures is that they support nesting. This means we can have data structures with data structures.

list1 = [1, 2, 3]

list2 = [4, 5, 6]

list3 = [7, 8, 9]

list_of_list = [list1, list2, list3]

print(list_of_list)

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

indexing

list_of_list[0][1]

o/p → 2.

Dictionaries

- Dictionary is a collection of key-value pairs.

A dictionary object is constructed by using curly braces

{ key1 : value1 , key2 : value2 , key3 : value3 }

```
my_diet = {'UP': 'LUCK', 'RAJ': 'JAI', 'PUN': 'CHO'}

my_diet['UP']

o/p → 'LUCK'

type(my_diet)

o/p → dict

my_diet['RAJ']

o/p → 'JAI'

my_diet['PUN']

o/p → 'CHO'
```

Dictionary method

keys()

↳ This method return the list of keys in the dictionary object.

```
my-dict.keys()
```

o/p → dict_keys (['UP', 'RAJ', 'PUN'])

```
list(my-dict.keys())
```

o/p → ['UP', 'RAJ', 'PUN']

values()

↳ This method returns the list of values in the dictionary object.

```
my-diet.values()
diet_values ([ ('LUCK', 'JAI', 'CHD'])
print( list(my-diet.values()))
```

o/p → ['LUCK', 'JAI', 'CHD']

items()

↳ This method returns the list of the key and values

list(my_dict.items())

o/p → [('UP', 'LUCK'), ('RAJ', 'JAI'), ('PUN', 'CHD')]

get()

↳ we can also use the get() method to extract a particular value of the key-value pair.

This method takes the key as an argument and returns None if the key is not found in the dictionary.

we can also set the value to return if key is not found.

This will be passed as the second argument in get()

my_dict = {"UP": "LUCK", "RAJ": "JAI"}

my_dict.get("UP")

o/p → "LUCK"

my_dict.get("HP")

o/p → None

my_dict.get("HP", "Not found")

o/p → Not found.

It is important to note that dictionaries are very flexible in the data types they can hold

my_dict = {"Name": "william", "list": [1,2,3]}

update() →

You can add an element which is key-value pair using the update method

This method takes a dictionary as an argument

```
my_dict.update({"college": "LBT MANDI"})
print(my_dict)
```

o/p → {'Name': 'william', 'list': [1,2,3,4],
"college": "LBT MANDI"}

⇒ we can also use update() method to update the existing value for a key.

```
my_dict.update({"Name": "Navodita"})
print(my_dict)
```

o/p → {'Name': 'Navodita', 'list': [1,2,3,4],
'college': 'LBT MANDI'}

dict()

↳ we can also create dictionary objects from sequence of items which are pairs. This is done using dict() function

This function takes the list of paired element as argument.

```
state_list = ['UP', 'HP', 'RAJ']

city_list = ['LUCK', 'SHM', 'JAI']

x = zip(state_list, city_list)

print(dict(x))
```

o/p → {'UP': 'LUCK', 'HP': 'SHM', 'RAJ': 'JAI'}

Python identation →

Identation refers to the space at the beginning of a code line.

Python uses identation to indicate a block of code.

What is flow control ?

↳ Flow control describe the order in which statements will be execute at runtime.

① conditional statement

- if
- if else
- else if

Syntax

```
if condition:
    statement
```

```
if condition:
    statement
else:
    statement
```

```
if condition:
    statement
elif condition:
    statement
else:
    statement
```

Example ①

```
user_input = input("enter a number")
user_input = int(user_input)

if user_input % 2 == 0:
    print("number is even")

else:
    print("number is odd")
```

example 2

```
age = int(input("enter your age"))

if age < 18:
    print("you are too young to marry")

elif age > 60:
    print("you are too old for marry")

else:
    print("we will find a perfect match for you")
```

→ Loops are important in Python or in any other programing language as they help you to execute a block of code repeatedly.

→ You will often come face to face with situations where you would need to use a piece of code over and over but you donot want to write the same line of code multiple times.

→ primarily there are two types of loops in Python

① while loop

② for loop

Advantage of loops

① Loops allow the execution statements or group of statment multiple times.

② In order to enter the loop there are certain condition defined in the biginning.

③ once the condition becomes false the loop stops and the control moves out of the loops.

For Loop

For loop in python is used to iterate over a sequence (that could be a list, tuple, string, or range of numbers)
it executes a set of statements for each each item in the sequence.

Syntax:

```
For item in object:
        statement
```

For loop on string →

```
my_sub = "Python"
for i in my_sub:
    print(i)
```

o/p→
P
y
t
h
o
n

For loop on range() function →

```
my_sub = "Python"
for i in range(len(my_sub)):
    print(my_sub[i], i)
```

o/p →

```
p 0
y 1
t 2
h 3
o 4
n 5
```

for loop on list →

```
my_list = [100, 200, 150, 300]
for i in my_list:
    print(i)
```

o/p

```
100
200
150
300
```

While loop

While loop is used to execute a block of code repeatedly until a given condition is satisfied.

```
i = 0          → initialization
while i<5 :    → condition
    print(i)
    i+=1       → increment
```

O/P →

```
0
1
2
3
4
```

Loop control statement

Break
- This statement break your loop.

continue
- This statement skip your current iteration

pass → This statement only pass your code.

```
# Break
i = 0
while i<6 :
    i+=1
    if i == 3 :
        break
        print(i)
    else :
        print(i)
```

o/p
1
2

continue

```
i=0
while i<6:
    i+=1
    if i==3:
        continue
        print(i)
    else:
        print(i)
```

O/P

1
2
4
5
6

Pass

```
i=0
while i<6:
    i+=1
    if i==3:
        pass
        print(i)
    else:
        print(i)
```

O/P →

1
2
3
4
5
6

User defined function

- User defined function allows you to define functions according to your need.
- These function are known as user-defined function.

user defined function without argument

```
def add():
    a = int(input("Enter first number: "))
    b = int(input("Enter second number: "))
    return a+b

add()
Enter first number: 5
Enter second number: 7
12
```

positional arguments →

↳ If a function has argument then we need to pass these value during call of function

→ positional arguments are the arguments that are passed to a function or method in a specific order.

→ The order in which you pass these arguments matters, because the function or method will use the arguments in the order that they are received.

For example
If you have a function that takes two arguments, the first arguments will be used as the first parameter, and the second argument will be used as the second parameter.

```
# positional arguments
def calculator(a, b):
    print("addition:", a+b)
    print("subtraction:", a-b)
    print("multiplication:", a*b)
    print("div", a/b)

calculator(6,3)
addition: 9
subtraction: 3
multiplication: 18
div: 2.0
```

Default arguments

Default arguments are arguments that are given a default value when the function or method is defined

These arguments are optional, because the function or method will use the default value if no value is provided for that argument when

the function called.

Default arguments can be used to provide default values for optional parameters, or to specify default behaviour for a function or method.

Python allows us to create functions with parameter that have default values.

```
def area_perimeter(width=2, height=2):
    area = width * height
    perimeter = (2*width) + (2*height)
    print("Area = " + str(area) + " and perimeter = "
          + str(perimeter))

area_perimeter()
```

Area = 4 and perimeter = 8

Keyword arguments

Keyword arguments are arguments that are passed to a function or method using the name of the arguments followed by an equal sign and the value of the argument.

These arguments do not need to be passed in a specific order, because the function or method will use the names of the arguments to determine which values to use for which parameters.

```
def interest(P, R, t):
    I = (P*R*t)/100
    return I

interest(R=10, t=2, P=1000)
```

o/p → 200.0

mixing Positional and keyword arguments

it is possible to mix positional arguments with keyword arguments But the positional arguments cannot appear after any keyword arguments have been defined

```
interest(1000, t=2, r=10)
200.0
```

Variable length arguments →

Variable length arguments are a powerful feature in Python that makes functions more flexible and adaptable.
it allows you to create functions that can accept any number of input values, making your code more efficient and user-friendly.

```
# Non keyword positional variable length arguments
def test(*args):
    print(args)
    print(len(args))
    print(type(args))
```

test(2, 3, 6, 9, 7, 5)

o/p→ (2, 3, 6, 9, 7, 5)

6

tuple

```
def sum_number(*args):
    sum = 0
    for sum in args:
        sum += num

    return sum
```

sum_number(2, 5, 7, 8)

o/p→ 22

keyword positional variable length arguments

```
def test(**kwargs):
    print(kwargs)
    print(len(kwargs))
    print(type(kwargs))
```

test(a=10, b=20, c=30, d=35)

{'a': 10, 'b': 20, 'c': 30, 'd': 35}

4

dict

```
def pass_students(**kwargs):
    pass_student_list = []
    for key, value in kwargs.item():
        if value > 33:
            pass_student_list.append(key)
    return pass_student_list

pass_students(Adam = 23, Ricky = 35, Symond = 45,
              babu = 67, rambabu = 48)
```

['Ricky', 'Symond', 'babu', 'rambabu']