'>>>' in python terminal is called 'Chevron Prompt'

docstrings should be the first line in the function, class modules, so on....., and has to be multi-line comments only. Not regular comments.

print(pythonobject.__doc__)

In [1]:
```
x = 5
y = "John"
print(type(x))
print(type(y))
```

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

In [2]:
```
x,y,z="Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

```
Orange
Banana
Cherry
```

In [3]:
```
x=y=z="Orange"
print(x)
print(y)
print(z)
```

```
Orange
Orange
Orange
```

In [4]:
```
Upfruits = ["apple", "banana", "guava"]
x,y,z=fruits
print(x)
print(y)
print(z)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-3de367f0824f> in <module>
      1 Upfruits = ["apple", "banana", "guava"]
----> 2 x,y,z=fruits
      3 print(x)
      4 print(y)
      5 print(z)

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

In [5]:

```
#RHS should always be a list or tuple

*x = 5,4,8,9,6,3,2,1,5,4,7
print(x)
```

```
  File "<ipython-input-5-d6da25905974>", line 3
    *x = 5,4,8,9,6,3,2,1,5,4,7
    ^
SyntaxError: starred assignment target must be in a list or tuple
```

In [6]:

```
fruits = ["apple", "banana", "guava"]
*x,y=fruits
print(x)    # whether its List or Tuple, *x will always be list natively
print(y)
```

```
['apple', 'banana']
guava
```

In [7]:

```
x="Sharan "
y="Jaiswal"

def myFunc():
    x="Saint "  # this local var will override the global var with the same name.
    global z    # For making the scope global, firstly declare the object as global. Then define it.
    z="A boy"
    global y
    y="jaiswaaaaaal"    # Not overriding the global var 'y' with local var 'y'. Instead we are accessing that global
    print(x+y)  #also, if accessing of global var is used for changing its value, then within that scope, it cant be

myFunc()

print(x+y)
print(z)
```

```
Saint jaiswaaaaaal
Sharan jaiswaaaaaal
A boy
```

`type(var) :`

Text Type: str

Numeric Types: int, float, complex

Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

Boolean Type: bool

Binary Types: bytes, bytearray, memoryview

In [8]:

```
x = 2
print(isinstance(x, int))

# If the type parameter is a tuple (any other iterable is not allowed), this function will return True if the object

x = isinstance("Hello", (float, int, str, list, dict, tuple))
print(x)


class myObj:
  name = "John"
y = myObj()
x = isinstance(y, myObj)
print(x)
```

```
True
True
True
```

In [9]:

```
import random
print(random.randrange(1, 10))
```

```
8
```

In [10]:

```
txt = "The best things in life are free!"
print("free" in txt)
if "free" in txt:
  print("Yes, 'free' is present.")
if "expensive" not in txt:
  print("Yes, 'expensive' is NOT present.")
```

```
True
Yes, 'free' is present.
Yes, 'expensive' is NOT present.
```

In [11]:
```
a = 2
b = 330

if a < b: print("a is greater than b")

print("A") if a > b else print("B")

a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

```
a is greater than b
B
=
```

In [12]:
```
for x in range(6):
  print(x)
 # if x == 5:
 #      break
  continue
else:
  print("Finally finished!")
# The else keyword in a for loop specifies a block of code to be executed when the loop is finished. The else block
# else can be used in while loops als o
# for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement t

for x in [0, 1, 2]:
  pass

# iteration variable can be more than 1 on for loop, generally used to iterate dict.items()
```

```
0
1
2
3
4
5
Finally finished!
```

Functions

In [13]:

```
def func(x):
    print(type(x))    # normal user defined function sends collection as a whole to the function
ages = [5, 12, 17, 18, 24, 32]
func(ages)
```

```
<class 'list'>
```

In [14]:

```
def my_function(*kids):  # Arbitrary Arguments, *args . add a * before the parameter name in the function definition
  print("The youngest child is " + kids[2])
  for name in kids:
    print(name)
  print(type(kids))

my_function("Emil", "Tobias", "Linus")

#####################################################

def my_function(child3, child2, child1):  # Keyword Arguments *kwargs . with known number of keyword arguments
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

#####################################################

def my_function(**kid):  # Arbritary Keyword Arguments **kwargs . with unknown number of keyword arguments
  print("His last name is " + kid["lname"])
  print(type(kid))

my_function(fname = "Tobias", lname = "Refsnes")

#####################################################

def hello(*args, **kwargs):
    print(args)
    print(kwargs)

# hello("Sharan", age = 24, "Jaiswal", dob = 1999) # It wont run. Hence order of parameters must be followed
hello("Sharan", "Jaiswal", age = 26, dob = 1999)

lst=["Sharan", "Jaiswal"]
dict_args = {'age': 26, 'dob':1999}
hello(lst, dict_args)    # this will transfer arguments as positional/arbritary args, and kwargs will get nothing
#rectified:
hello(*lst, **dict_args)
# hello(**dict_args, *lst) # Error. Even now also, arguments order cant be changed wrt parameters
```

```
The youngest child is Linus
Emil
```

```
Tobias
Linus
<class 'tuple'>
The youngest child is Linus
His last name is Refsnes
<class 'dict'>
('Sharan', 'Jaiswal')
{'age': 26, 'dob': 1999}
(['Sharan', 'Jaiswal'], {'age': 26, 'dob': 1999})
{}
('Sharan', 'Jaiswal')
{'age': 26, 'dob': 1999}
```

In [15]:

```python
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("India")
my_function()

#####################################################

def my_function(food):  # Passing a list as an arguments
  for x in food:
    print(x)
  print(type(food))

fruits = ["apple", "banana", "cherry"]

my_function(fruits)

#####################################################

def myfunction():  # when no function definition is present
  pass

#####################################################

def tri_recursion(k):
    if (k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
        print(result)
    return result

tri_recursion(3)
```

```
I am from India
I am from Norway
apple
banana
cherry
<class 'list'>
```

```
0
1
3
```

```
Out[15]: 6
```

```
In [16]: def evenoddsum(lst):
             esum=0
             osum=0
             for i in lst:
                 if (i%2==0):
                     esum=esum+i
                 else:
                     osum=osum+i
             return esum, osum    # multiple returns are always items of a tuple

         lst = [1,2,3,4,5,6,7,8,9,0]
         print(evenoddsum(lst))
```

```
(20, 25)
```

Lambda

anonymous function; one or many argsuments ===> one return value

In [17]:

```
x = lambda a, b : a * b +10
print(x(5, 6))

def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))
print(mytripler(11))

def build_quadratic_function(a,b,c):
    """Returns ther function f(x) = ax^2 + bx + c"""
    return lambda x : a*x**2 + b*x + c
f = build_quadratic_function(2, 3, -5)
print(f(0))
print(f(1))
print(f(2))
build_quadratic_function(3, 0, 1)(2)  # 3x^2+1 evaluated for x=2
```

```
40
22
33
-5
0
9
```

Out[17]:

```
13
```

Exception Handling

Errors are composed of SYNTAX ERRORS & EXCEPTIONS(caught during run-time).

try: ... except: ... [else: ...][finally: ...]

1. Test block of code for errors
2. handle the error, You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error. Or, just you can send the error message to the user. You can use here **quit()** function to exit the program.

3. this block is defined with code, which will be executed when no errors is raised.
4. executes code, regardless of the result of try-except block. Generally used for cleanup operations.

In custom error message handling, we write the possible child error message at the top in the exception ladder. At last, we write the exception of the 'Exception' class

In [18]:

```
try:
  print(jabba)
except NameError:  # as ne # Print one message if the try block raises a NameError and another for other errors
  print("Variable jabba is not defined")
except:
  print("Something else went wrong")
```

```
Variable jabba is not defined
```

In [19]:

```
try:
    a=1
    b='s'
    c=a+b
except NameError as ex1:
    print("The user have not defined the variable")
except Exception as ex:
    print(ex)    # or here also one can put their custom message
```

```
unsupported operand type(s) for +: 'int' and 'str'
```

In [20]:

```
try:
    a=1
    b='s'
    c=a+b
except NameError:
    print("The user have not defined the variable")
except TypeError:
    print("Try to make datatype similar")
except Exception as ex:
    print(ex)    # or here also one can put their custom message
```

```
Try to make datatype similar
```

In [21]:
```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

```
Hello
Nothing went wrong
```

In [22]:
```
#del y # toggle this line as a comment to see the effect
try:
  print(y)
except:
  print("Something went wrong")
else:
  print("try didnt gets executed")
finally:
  print("The 'try except' is finished")
```

```
<__main__.myObj object at 0x7f8b1c3cba90>
try didnt gets executed
The 'try except' is finished
```

Raise/Throw an exception

An exception can be raised if certain user-defined condition occurs. You can define what kind of error to raise, and the text to print to the user.

Exception

This is the main exception class. All the other exception class are derived from this exception class.

In [23]:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")

# code will not execute after this, even though error is raised and handeled
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-23-1129764d2a7a> in <module>
      2
      3 if x < 0:
----> 4   raise Exception("Sorry, no numbers below zero")
      5
      6 # code will not execute after this, even though error is raised and handeled

Exception: Sorry, no numbers below zero
```

In [24]:

```
p=q
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-af532074f79a> in <module>
----> 1 p=q

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

This above specific type of exception, i.e., 'NameError' exception is basically derived from the 'Exception' class.

In [25]:

```
try:
    p=q
except Exception as ex:    # here we called the Exception Class and aliased it as 'ex'
    print(ex)    # here we can see that the ex contains the same error message string that was given in the above ce
```

```
name 'q' is not defined
```

One can handle this exception and customize this exception message. This is called exception handling, and we did it in the previous section.

In [26]:

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-ac71689f5629> in <module>
      2 
      3 if not type(x) is int:
----> 4   raise TypeError("Only integers are allowed")

TypeError: Only integers are allowed
```

In [27]:

```
x = "hello"
# if error/exception is raised in try block, then it can be handeled
if not type(x) is int:
    try:
        raise TypeError("Only integers are allowed")
    except:
        print('handled')
```

```
handled
```

Creating Custom Exception

In [67]:
```
class Error(Exception):    # deriving Exception class into custom generic class.
    pass    # instead defining any particular exception, we'll print our own exception

class dobException(Error):
    pass

year=int(input("Enter the year of birth: "))
age=2021-year
try:
    if age<=30 and age>20:
        pass
    else:
        raise dobException
except dobException:
    print("age out of range")
```

```
Enter the year of birth: 2015
age out of range
```

LEARN COMMON ERROR TYPES AND THEIR PROCESS TO RAISE ERROR TYPE AND THEIR HANDLING

Built-in Function of python

abs(\<num>)

all(\<iterable>) # returns True if all the elements from the iterable are True. Else False.

any(\<iterable>)

bin(\<num>) # return binary number of integer \<num>. Other numbers are not allowed

bool() # It will return False if parameters will be empty objects like empty list, dictionary, tuple, string, 0, 0.00 ..., False, None

complex([']real, [imaginary number pary w/o 'j'][']) # outputs (r+ij)

filter(bool returning function for each item in iterable, iterable) # returns an iterator where the items are filtered through a function to test if the item is accepted or not.

min() max()

ord() . It returns the number representing the unicode code of a specified character.

chr() . returns the character that represents the specified unicode.

pow(base, exponent[, modulus]) # modulus is always any integer except 0, allowed only when base and exponent are integer.

hash() Returns the hash value of a specified object

FILTER Function

callable(object_name) Returns True/False

In [29]:
```python
def myFunc(x):
    if x < 18:
        print(type(x))    # filter function sends one by one colletion items. NOT whole item at a time, as a normal
        return False
    else:
        return True

# filter only looks for true value. not false in general. Above AKA: filter(lambda num: num<18, ages)

ages = [5, 12, 17, 18, 24, 32]
adults = filter(myFunc, ages)
print(adults)
#print(list(adults))
for x in adults:
    print(x)
```

```
<filter object at 0x7f8b1c3802b0>
<class 'int'>
<class 'int'>
<class 'int'>
18
24
32
```

MAP Function

In [30]:

```python
def myfunc(a, b):
  return a + b
x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))
print(x)    # returns map object. Memory has not been instantiated using map. Because it used lazy-loading technique
#convert the map into a list, or in any collection form, for readability:
print(list(x))


# map() function executes a specified function for each item in an iterable. The item is sent to the function as a p
# function: Required. The function to execute for each item
#iterable: Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just
```

```
<map object at 0x7f8b1c3660a0>
['appleorange', 'bananalemon', 'cherrypineapple']
```

ZIP Function

In [31]:

```python
a = ("John", "Charles", "Mike")
b = ["Jenny", "Christy", "Monica", "Vicky"]
x = zip(a, b)    # input could be be any iterable. Output is iterator.
print(x)
print(list(x))
print(tuple(x))
print(x)

# returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired togeth
```

```
<zip object at 0x7f8b1c336380>
[('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica')]
()
<zip object at 0x7f8b1c336380>
```

In [32]:
```
a = ("John", "Charles", "Mike")
b = ["Jenny", "Christy", "Monica", "Vicky"]
x = zip(a, b)    # input could be be any iterable

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

```
('John', 'Jenny')
('Charles', 'Christy')
('Mike', 'Monica')
```

In [33]:
```
a = ("John", "Charles", "Mike")
b = ["Jenny", "Christy", "Monica", "Vicky"]
x = zip(a, b)    # input could be be any iterable

for i,j in x:
    print(i, j)
```

```
John Jenny
Charles Christy
Mike Monica
```

In [34]:
```
# creating zip object
list(zip()) # one can put tuples as zip items inside zip()
```

Out[34]:
```
[]
```

In [35]:
```
dict1={'name':"Sharan", "title":"jaiswal", "age": 23}
dict2={'name':"jaiswal", "title":"sharan", "age": 32}
```

In [36]:
```
dictionary = zip(dict1, dict2)
```

In [37]:
```
for i in dictionary:
    print(i)
```

```
('name', 'name')
```

```
('title', 'title')
```

In [38]:
```
dictionary = zip(dict1.items(), dict2.items())
```

In [39]:
```
for i in dictionary:
    print(i)
```

```
(('name', 'Sharan'), ('name', 'jaiswal'))
(('title', 'jaiswal'), ('title', 'sharan'))
(('age', 23), ('age', 32))
```

In [40]:
```
for (i,j),(i2, j2) in dictionary:
    print(i,j)
    print(i2, j2)
```

EVAL Function

VALUATING EXPRESSIONS DYNAMICALLY. Evaluates python expression which are written as strings.

1. Parse the expression.
2. Compile the expression into byte code.
3. Evaluate the python expression.
4. Return the result.

eval(source, globals=None, locals=None, /)

In [41]:
```
eval("5*5+7/2")
```

Out[41]:
```
28.5
```

In [64]:
```
eval(input("Enter the expression to evaluate : "))
```

```
Enter the expression to evaluate : 5*8/6
```

```
Out[64]: 6.666666666666667
```

```
In [43]: def sq_num(num):
             return num**2
```

```
In [44]: sq_num(2)
```

```
Out[44]: 4
```

```
In [45]: eval("sq_num(2)")
```

```
Out[45]: 4
```

working of eval. compile()

```
In [46]: var=compile("5*5", "<string>", "eval")   # parsing and compiling into the byte code
```

```
In [47]: var
```

```
Out[47]: <code object <module> at 0x7f8b1c432870, file "<string>", line 1>
```

```
In [48]: eval(var)
```

```
Out[48]: 25
```

Globals parameter

```
In [49]: eval("x+50+x**2")    # it will give error because x is not defined
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-40ee5cd58b41> in <module>
----> 1 eval("x+50+x**2")    # it will give error because x is not defined

<string> in <module>
```

```
TypeError: unsupported operand type(s) for +: 'zip' and 'int'
```

In [50]:
```
eval("mu+50", {"mu": 10})
```

Out[50]:
```
60
```

In [51]:
```
x=20
eval("x+50+x**2", {"x":x})
```

Out[51]:
```
470
```

In [52]:
```
x=0
eval("x+50+x**2")
```

Out[52]:
```
50
```

In [53]:
```
w=34
x=100
y=20
z=100
eval("w+x+y+z", {"x":x, "z":z, "w": 40})
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-53-d49df39fbe2f> in <module>
      3 y=20
      4 z=100
----> 5 eval("w+x+y+z", {"x":x, "z":z, "w": 40})

<string> in <module>

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

In [54]:
```
eval("w+x+y+z", {}, {"x":1, "z":2, "w": 3, "y":4})    # locals
```

Out[54]:
```
10
```

Minimizing security issues

Sorting and Compairing of collection items

Any sequences can be compared. Underlying concept is, corresponding elements are compared.

One can use **sorted(iterable, key=key, reverse=True/False)** function. Sorting of collection cant be done when that sequence contains BOTH string values AND numeric values. Strings are sorted alphabetically, and numbers are sorted numerically.

In [55]:
```
lst = [0,1,2,'a', "sharan", 'b', "Sharan", None]
x = sorted(lst)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-1e5348b9b154> in <module>
      1 lst = [0,1,2,'a', "sharan", 'b', "Sharan", None]
----> 2 x = sorted(lst)

TypeError: '<' not supported between instances of 'str' and 'int'
```

In [56]:
```
lst=[0,1,2,3,4]
x = sorted(lst)
x
```

Out[56]:
```
[0, 1, 2, 3, 4]
```

In [57]:
```
lst=['a', 'b', 'c', 'A', "sharan", "SHARAN", 'B', 'C', 'd', 'e', '0','1','6','9']
x = sorted(lst)
x
```

Out[57]:
```
['0',
 '1',
 '6',
 '9',
 'A',
 'B',
 'C',
 'SHARAN',
 'a',
 'b',
 'c',
 'd',
 'e',
 'sharan']
```

In [58]:

```
d={'b':10, 'c':1, 'a':22}
t=sorted(d.items())
t
```

Out[58]:

```
[('a', 22), ('b', 10), ('c', 1)]
```

In [59]:

```
# sorting based on values of dictionary using SORTED()

d={'b':10, 'c':1, 'a':22}
tmp=list()

for k,v in d.items():
    tmp.append((v,k))

print(tmp)

tmp=sorted(tmp, reverse = True)
print(tmp)


print(sorted([(v,k) for (k,v) in d.items()], reverse = True))
```

```
[(10, 'b'), (1, 'c'), (22, 'a')]
[(22, 'a'), (10, 'b'), (1, 'c')]
[(22, 'a'), (10, 'b'), (1, 'c')]
```

Assert Statements in Python

It is used to check if a given logical expression is True or False. Program execution proceeds only if the expressions is true and raises the **AssertionError** when it is False.

In [60]:
```
10>=10 # this is not assert statement
```

Out[60]:
```
True
```

In [61]:
```
10>10  # this is not assert statement
```

Out[61]:
```
False
```

In [62]:
```
num=12
assert num<10
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-62-cd3ba8613112> in <module>
      1 num=12
----> 2 assert num<10

AssertionError: 
```

In [63]:
```
try:
    num=int(input("enter a number: "))
    assert num%2==0
    print("The number is even.")
except AssertionError:
    print("Please enter even number")
```

```
enter a number: 456
The number is even.
```