PYTHON 3

BASIC SYNTAX

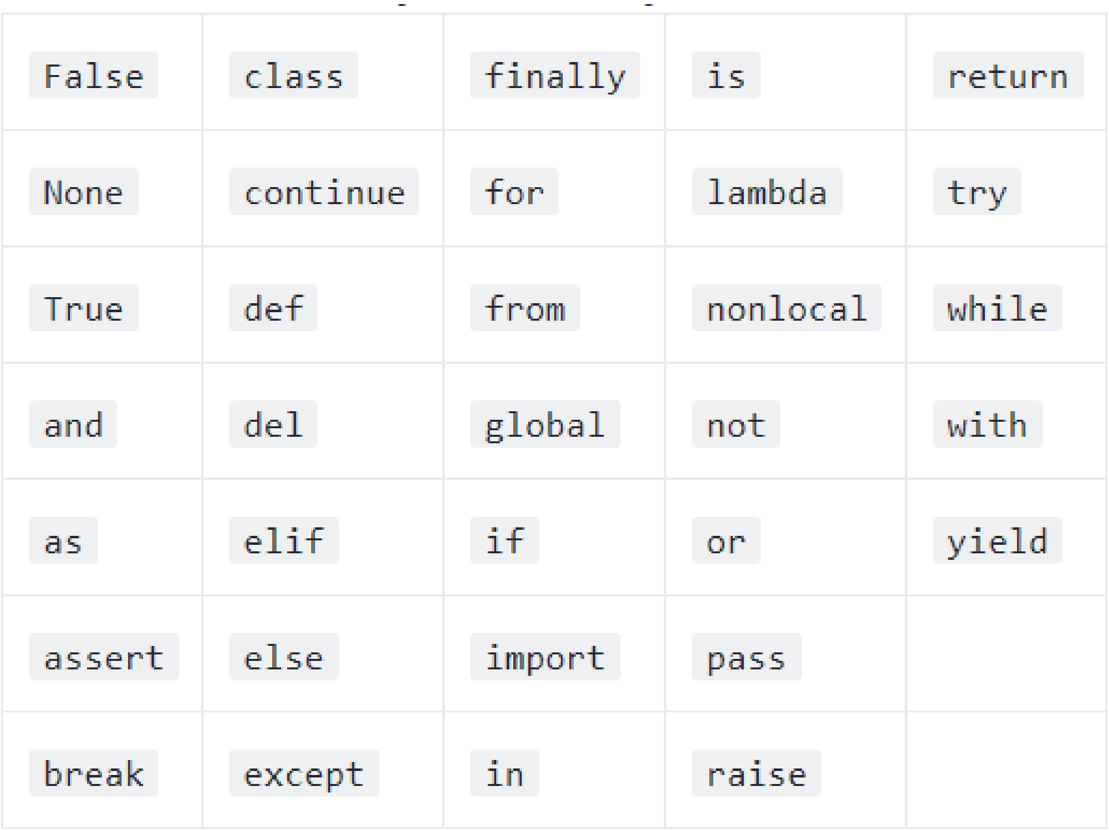
Python is an interpreted language

- You can write programs interactively using the interpreter

- You can also write scripts
  - File extention will be .py [eg. demo.py]
  - In console the script can be run by python command

Python Basic Concepts

- Identifier
- Reserved Words
  - 33 keywords

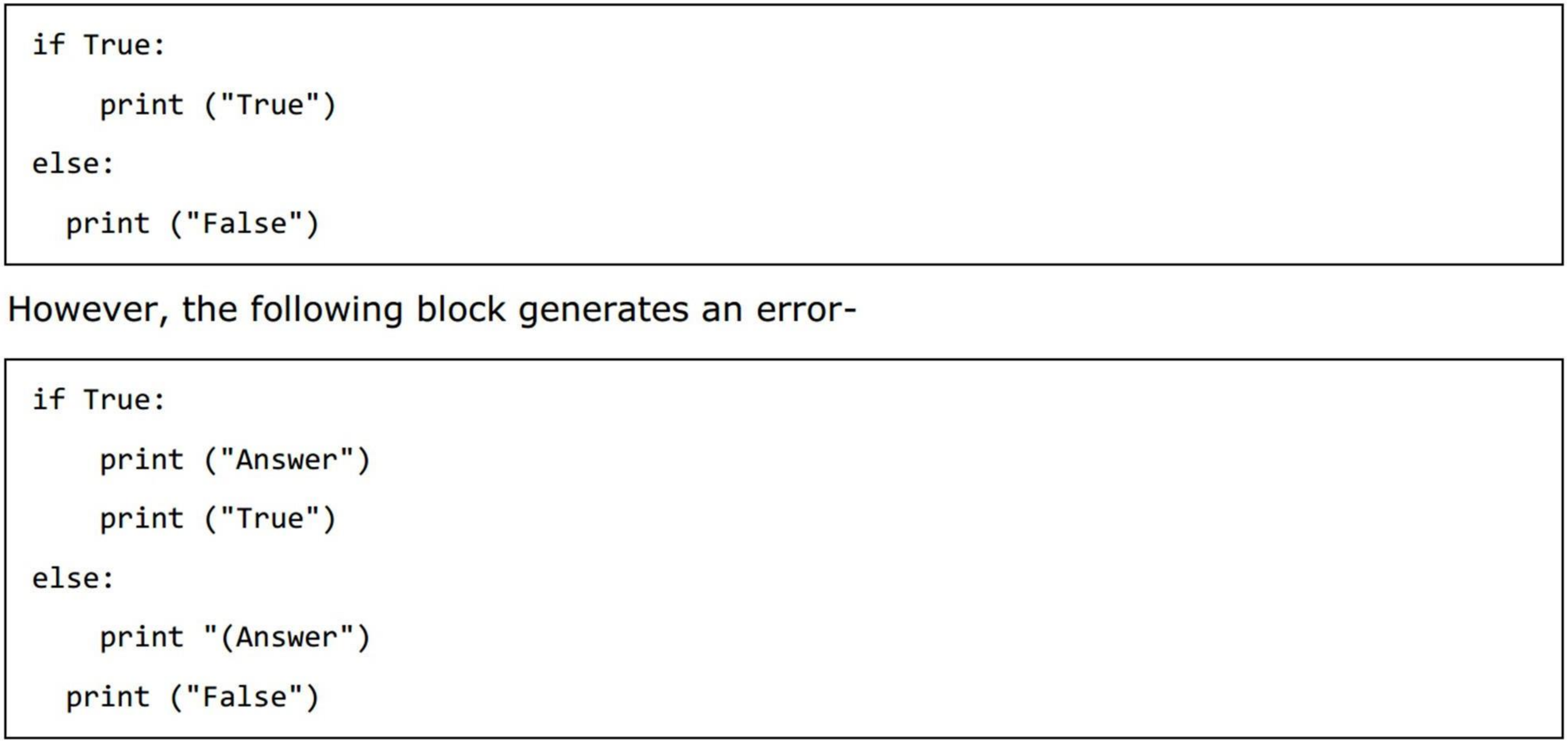


False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Lines and Indentation

- No semicolon needed at the end of lines
- Python does not use braces({}) to indicate blocks of code
- Blocks of code are denoted by line indentation
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.
- A single code block is also called **suites** in Python

Indentation



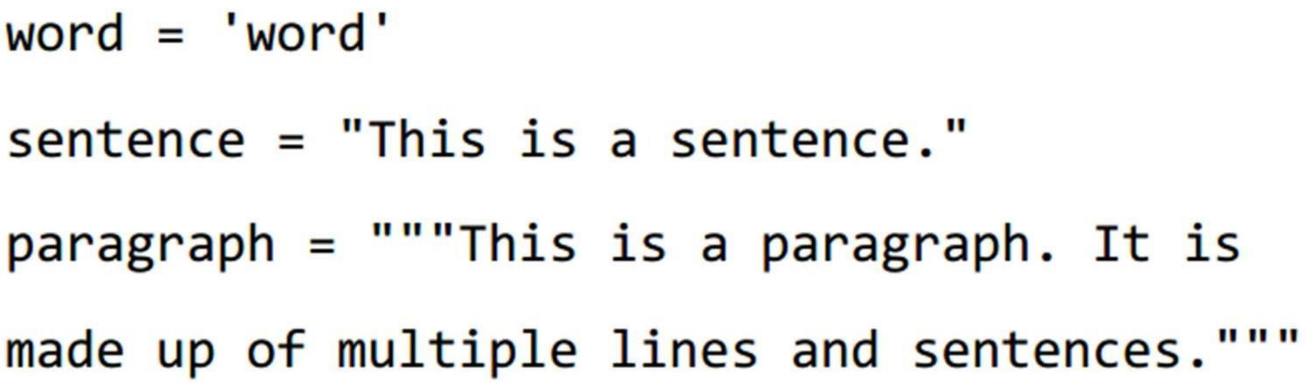
```
if True:
    print ("True")
else:
  print ("False")
```

However, the following block generates an error-

```
if True:
    print ("Answer")
    print ("True")
else:
    print "(Answer")
  print ("False")
```

Quotation in Python

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals
  - the same type of quote must start and end the string.
  - The triple quotes are used to span the string across multiple lines.



```
word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""
```

Comments

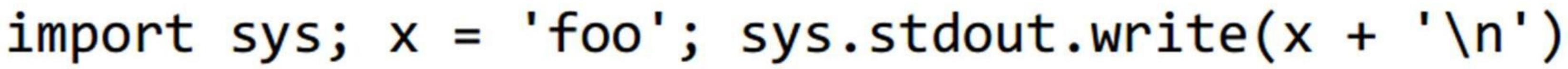
- Single line comment: #comment
- Triple quotes can be utilized for multiple-line commenting.

User Input

- input()
  - takes the next line from console
- input("\n\nPress the enter key to exit.")
- By default, input is a string.
- n = int(input()) # casts to int

Multiple Statements on a Single Line

- The semicolon (;) allows multiple statements on a single line



```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Print

- print(“String”, end = ‘’)#doesnt print \n after string

Multiple Assignment

- Python allows you to assign a single value to several variables simultaneously

- a = b = c = 1
- a, b, c = 1, 2, "john"

Variable Types

Standard Data Types

- Python has five standard data types-
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary

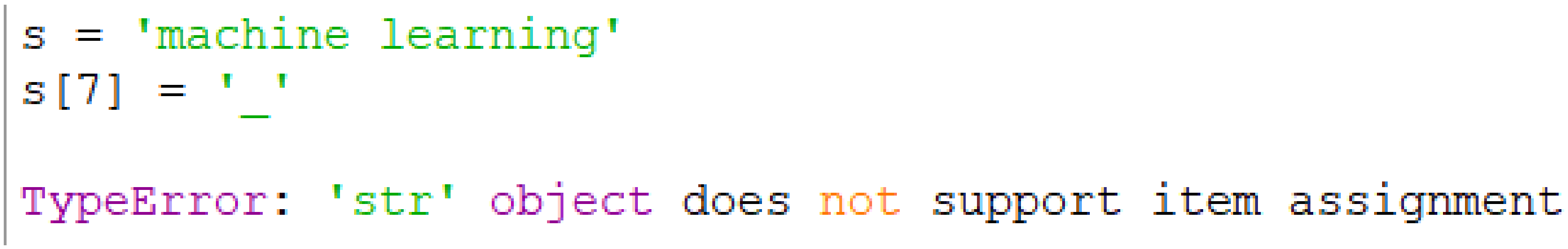
- No data type for characters
  - A character is just a string of length 1
- To find out the type of a object: type(var)

Numerical Types

- Python supports three different numerical types –
- int (signed integers)
  - You can store arbitrary large values
- float (floating point real values)
- complex (complex numbers)
  - A complex number consists of an ordered pair x + yj, where x and y are real numbers and j is the imaginary unit.

Strings

- A contiguous set of characters represented in the quotation marks.
- Python allows either pair of single or double quotes.
- It also has a multiline triple quote ‘‘‘‘‘‘ STRING ’’’’’’
- Strings are immutable in Python.



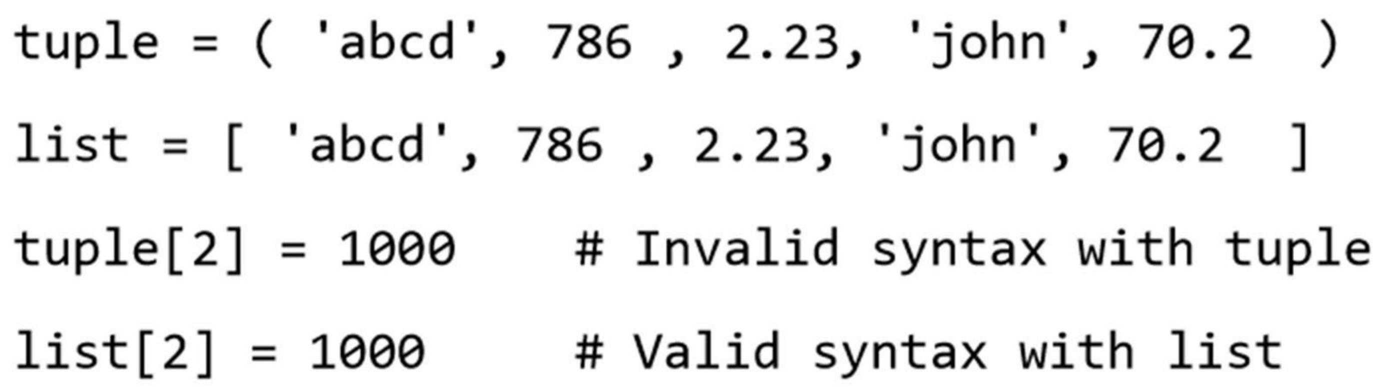
```
s = 'machine learning'
s[7] = '_'

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

Lists

- Most versatile among the compound data types
- A list contains items separated by commas and enclosed within square brackets ([1,2, “String”, ‘a’])
- Mostly like C arrays, however can contain items of different types

Python Tuples

- A tuple consists of a number of values separated by commas and enclosed within parenthesis.
- Unlike List Tuples can not be updated.
  - They are read only

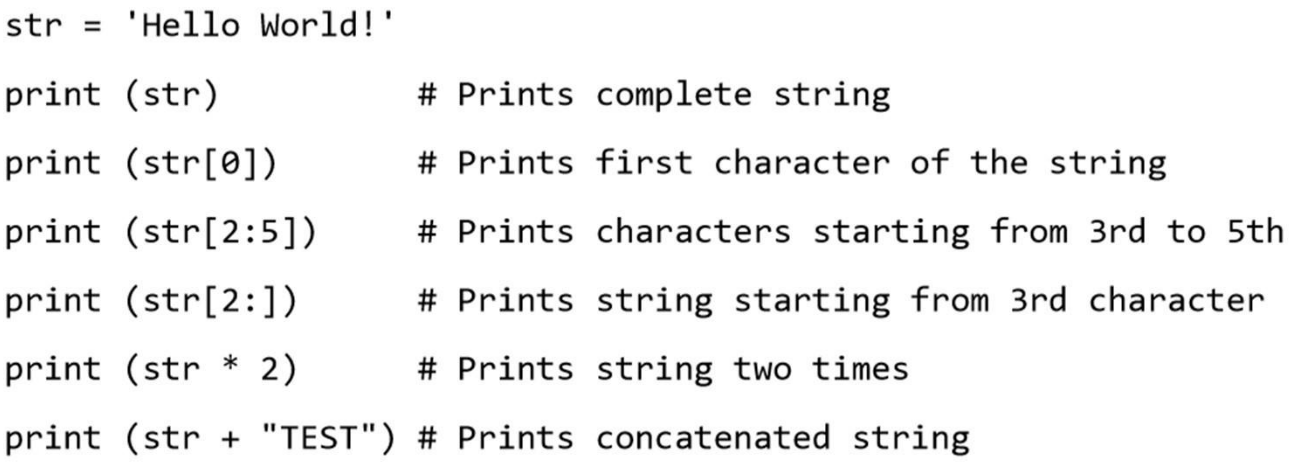


```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

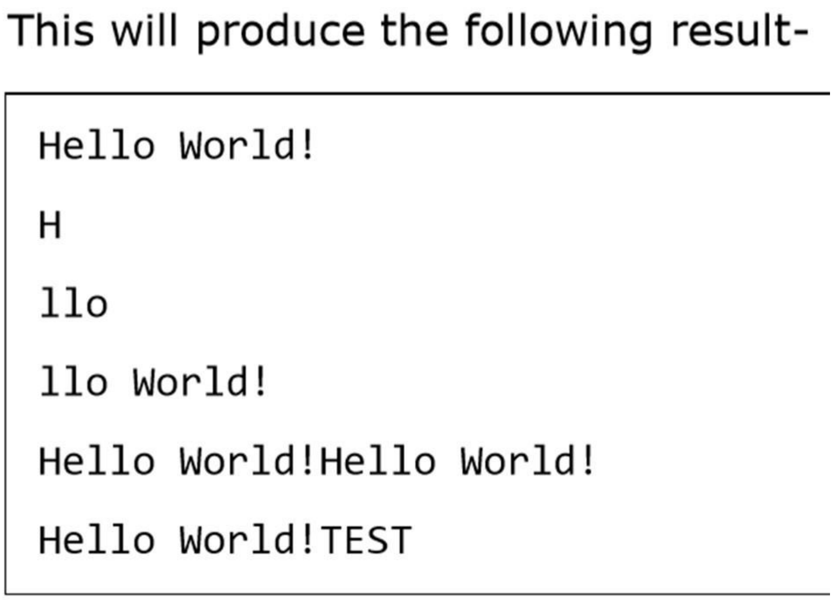
Common Operations/Functions on List, String, and Tuple

- Slicing: To get substrings, subLists ,or a single element the slice operator ([ ] and [:]) is used
  - indexes starts at 0 in the beginning
  - [inclusive:exclusive]
- The plus (+) sign is the concatenation operator
- The asterisk (*) is the repetition operator
- len() function returns the length

Example



```
str = 'Hello World!'
print (str)          # Prints complete string
print (str[0])       # Prints first character of the string
print (str[2:5])     # Prints characters starting from 3rd to 5th
print (str[2:])      # Prints string starting from 3rd character
print (str * 2)      # Prints string two times
print (str + "TEST") # Prints concatenated string
```



This will produce the following result-

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Example

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print (list)          # Prints complete list
print (list[0])       # Prints first element of the list
print (list[1:3])     # Prints elements starting from 2nd till 3rd
print (list[2:])      # Prints elements starting from 3rd element
print (tinylist * 2)  # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

Output

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Example

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
print (tuple)            # Prints complete tuple
print (tuple[0])         # Prints first element of the tuple
print (tuple[1:3])       # Prints elements starting from 2nd till 3rd
print (tuple[2:])        # Prints elements starting from 3rd element
print (tinytuple * 2)    # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

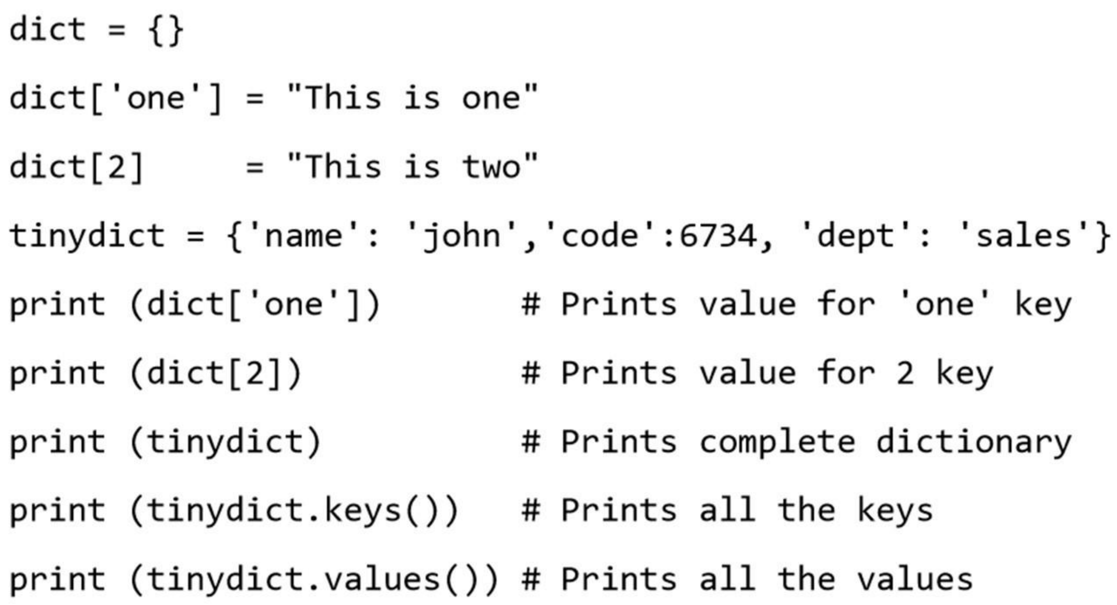
Output

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

Python Dictionary

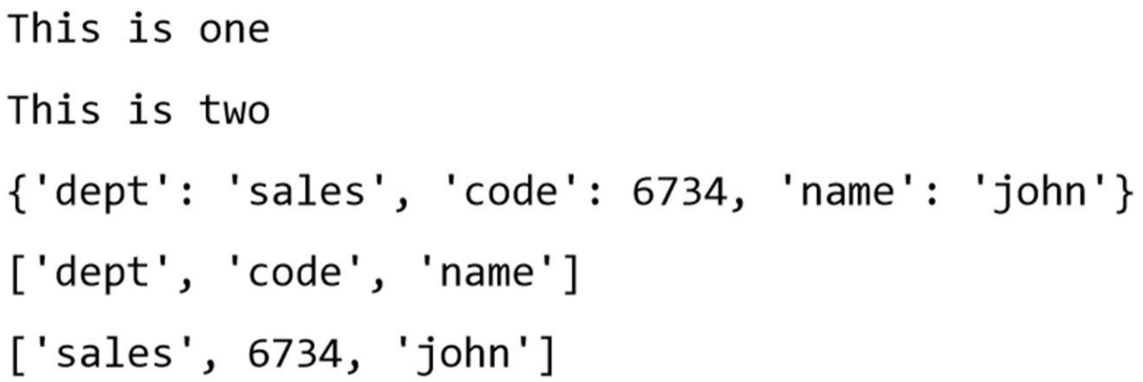
- Dictionaries can hold key-value pairs.
  - Similar to Map
  - A dictionary key can be almost any Python type, but are usually numbers or strings.
  - Values, on the other hand, can be any arbitrary Python object
  - Have no notion of order in data
- Dictionaries are enclosed by curly braces ({ })
- Values can be assigned and accessed using square braces ([])

Example



```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one'])       # Prints value for 'one' key
print (dict[2])           # Prints value for 2 key
print (tinydict)          # Prints complete dictionary
print (tinydict.keys())   # Prints all the keys
print (tinydict.values()) # Prints all the values
```

Output



```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Data Type Conversion

- To convert between the built-in types, simply use the type-name as a function.
- int(x [,base])
  - Converts x to an integer. The base (optional) specifies the base if x is a string.
- float(x), complex(real [,imag]), str(), chr()
- tuple(), list(), dict(), set()

BASIC OPERATORS

Operator Types

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

- Most are similar to C/Java
  - except Logical Operators

Arithmetic Operators

- + - * / %
- ** : power/ exponent
  - 3**2 ==9
- // : integer/floor division
  - 9//2 = 4, 9.0//2.0 =4.0
- no x++ or x--

Comparison Operators

- ==
- !=
- >
- <
- >=
- <=

Assignment Operators

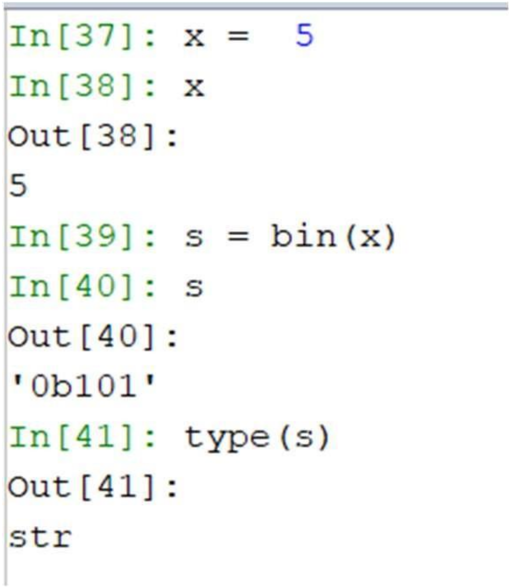
- =
- +=
- -=
- *=
- /=
- %=
- **=
- //=

Bitwise Operators

- &
- |
- ^
- ~
- <<
- >>

Bitwise Operators

- bin()
  - used to obtain binary representation of an integer number.



```
In[37]: x =  5
In[38]: x
Out[38]:
5
In[39]: s = bin(x)
In[40]: s
Out[40]:
'0b101'
In[41]: type(s)
Out[41]:
str
```

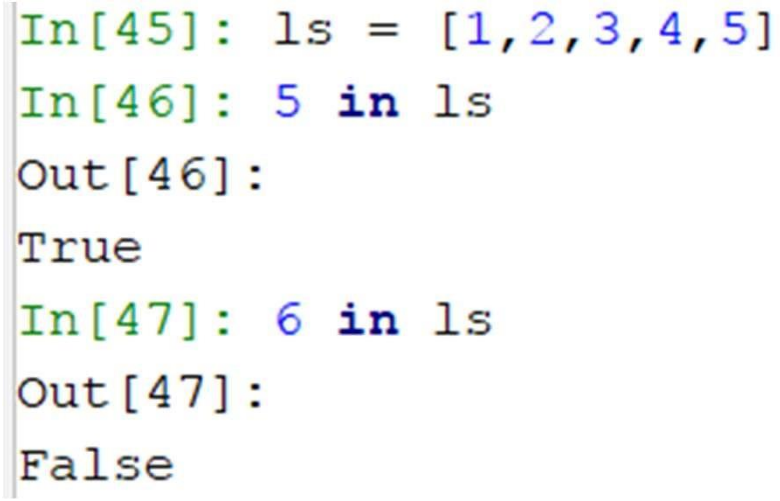
Logical Operators

- and
- or
- not

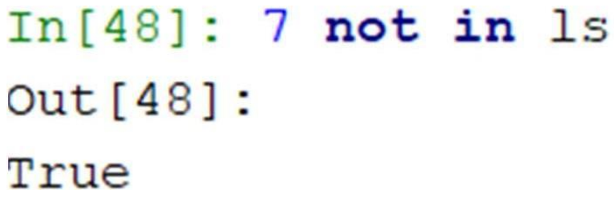
- These operators are UNLIKE C, C++ or Java

Python Membership Operators

- Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples.
- *in*
- *not in*



```
In[45]: ls = [1,2,3,4,5]
In[46]: 5 in ls
Out[46]:
True
In[47]: 6 in ls
Out[47]:
False
```

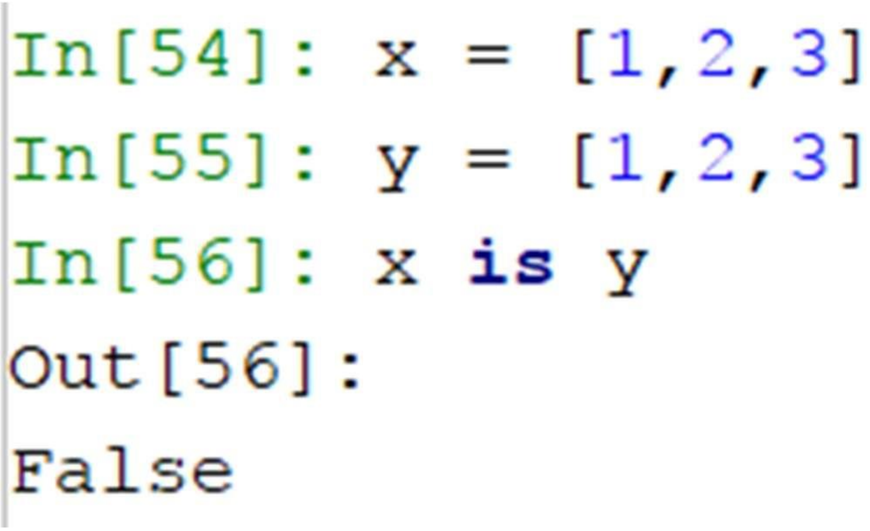


```
In[48]: 7 not in ls
Out[48]:
True
```

Python Identity Operators

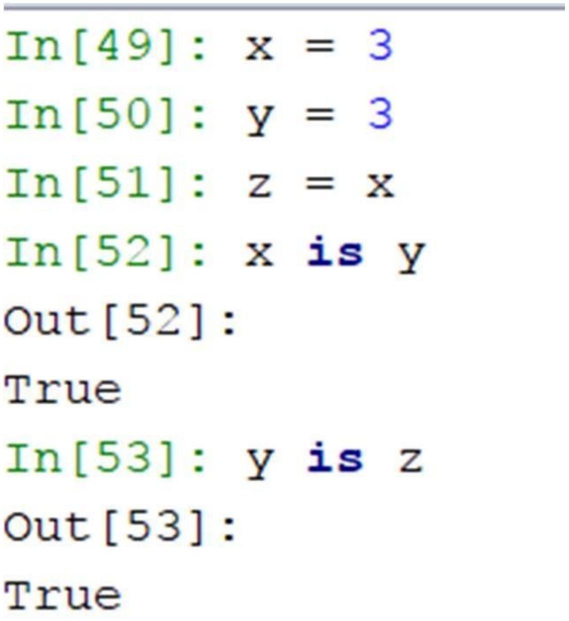
- Identity operators compare the memory locations of two objects

- is
- not is

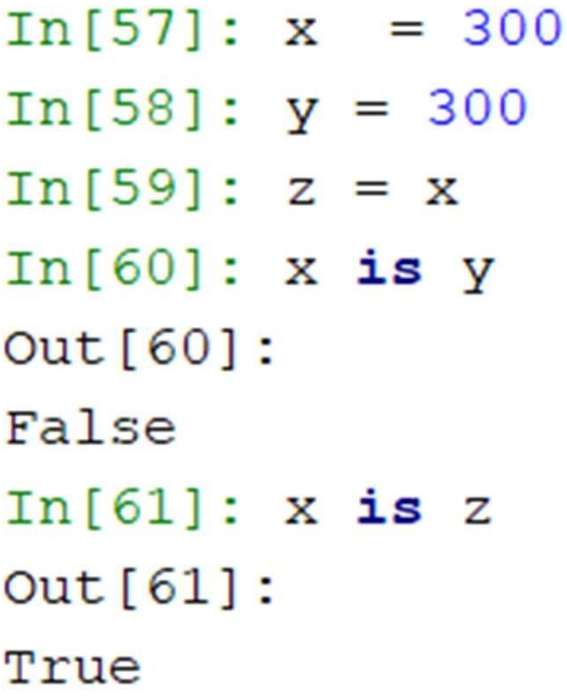


```
In[54]: x = [1,2,3]
In[55]: y = [1,2,3]
In[56]: x is y
Out[56]:
False
```

Python Identity Operators

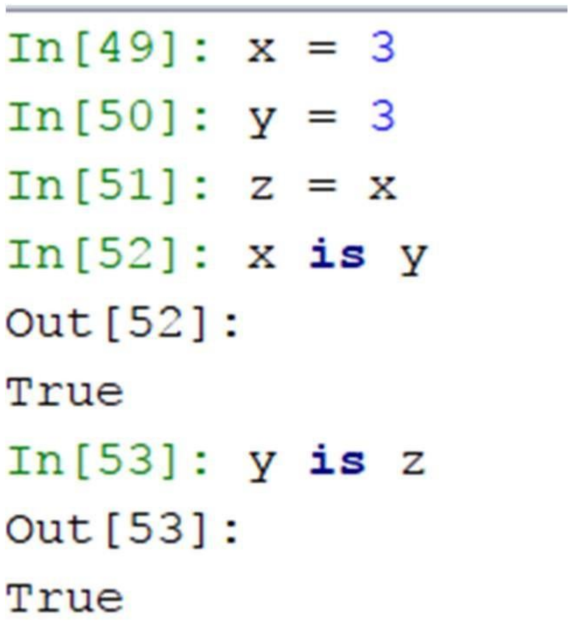


```
In[49]: x = 3
In[50]: y = 3
In[51]: z = x
In[52]: x is y
Out[52]:
True
In[53]: y is z
Out[53]:
True
```

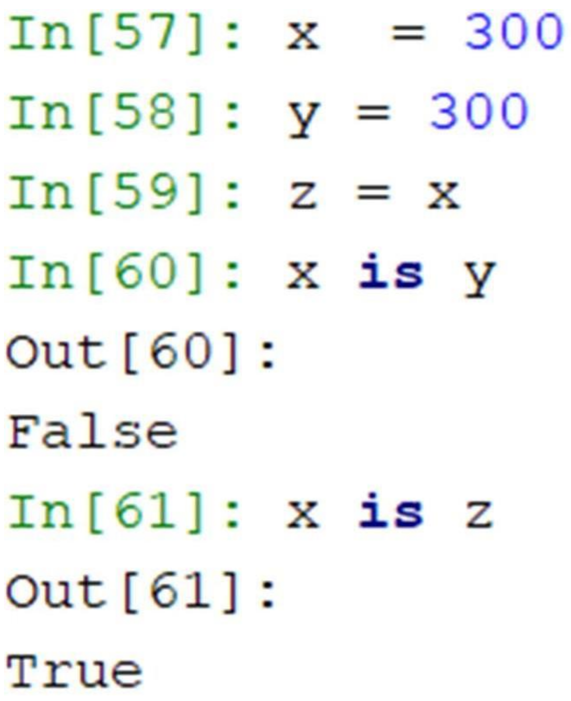


```
In[57]: x  = 300
In[58]: y = 300
In[59]: z = x
In[60]: x is y
Out[60]:
False
In[61]: x is z
Out[61]:
True
```

Python Identity Operators



```
In[49]: x = 3
In[50]: y = 3
In[51]: z = x
In[52]: x is y
Out[52]:
True
In[53]: y is z
Out[53]:
True
```

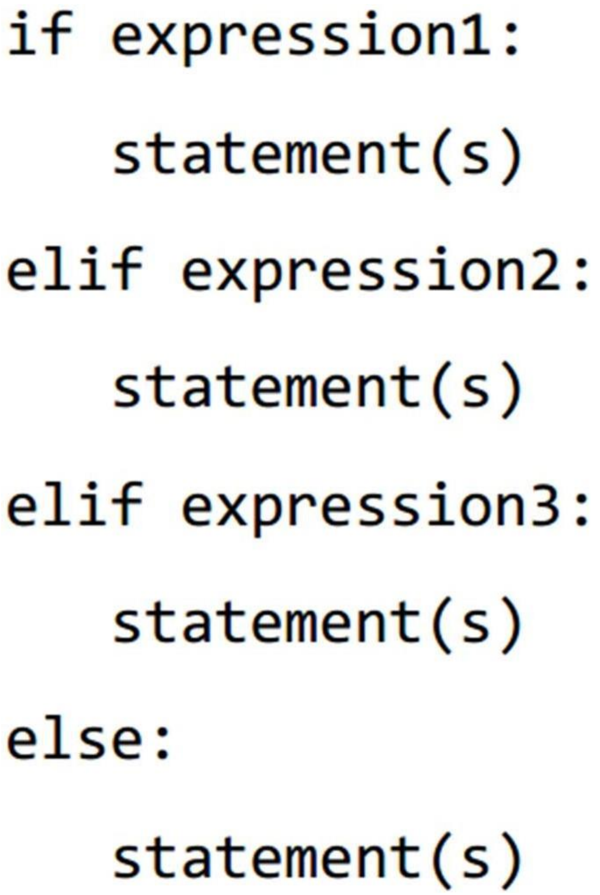


```
In[57]: x  = 300
In[58]: y = 300
In[59]: z = x
In[60]: x is y
Out[60]:
False
In[61]: x is z
Out[61]:
True
```

The current implementation keeps an array of integer objects for all integers between -5 and 256, when you create an int in that range you actually just get back a reference to the existing object.

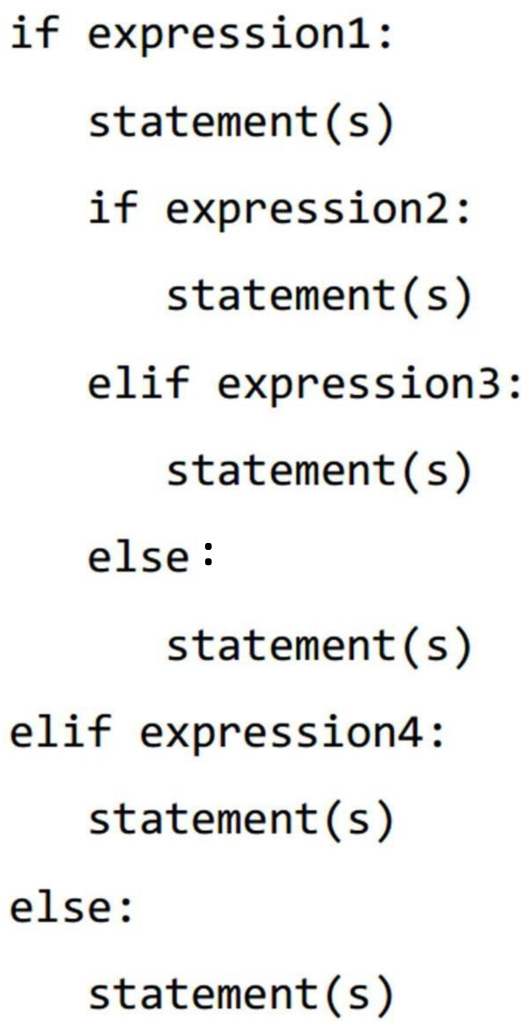
CONDITIONAL STATEMENTS

If - Else



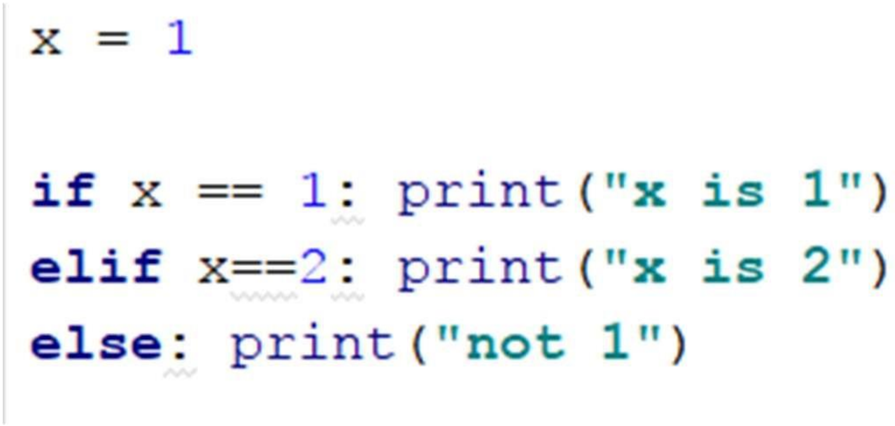
```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

Nested If



```
if expression1:
    statement(s)
    if expression2:
        statement(s)
    elif expression3:
        statement(s)
    else:
        statement(s)
elif expression4:
    statement(s)
else:
    statement(s)
```

Single Line If-Else

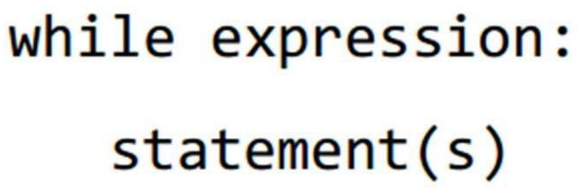


```
x = 1

if x == 1: print("x is 1")
elif x==2: print("x is 2")
else: print("not 1")
```

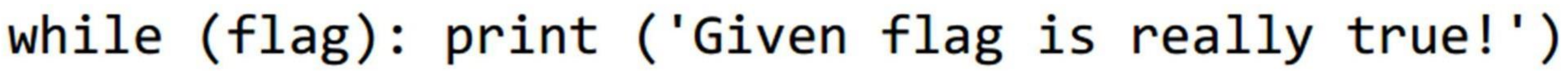
LOOPS

Loops

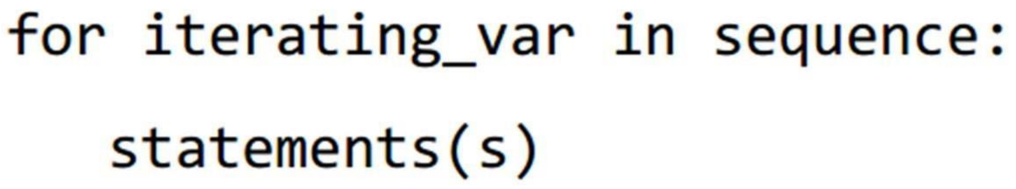
- while



```
while expression:
   statement(s)
```



```
while (flag): print ('Given flag is really true!')
```



```
for iterating_var in sequence:
   statements(s)
```

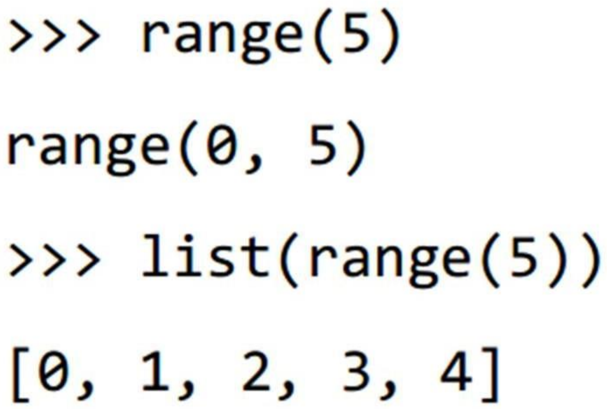
Range

- The built-in function range() is used to iterate over a sequence of numbers.

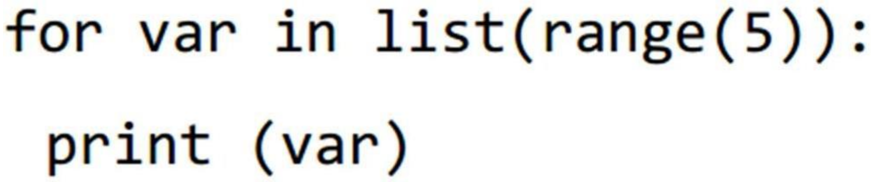
- range() generates an iterator to progress integers starting with 0 upto n-1
  - memory efficient

- To obtain a list object of the sequence, it is typecasted to list()

Range

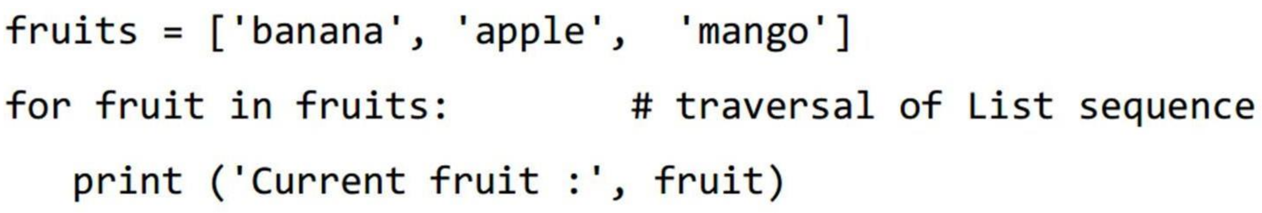


```
>>> range(5)

range(0, 5)

>>> list(range(5))

[0, 1, 2, 3, 4]
```

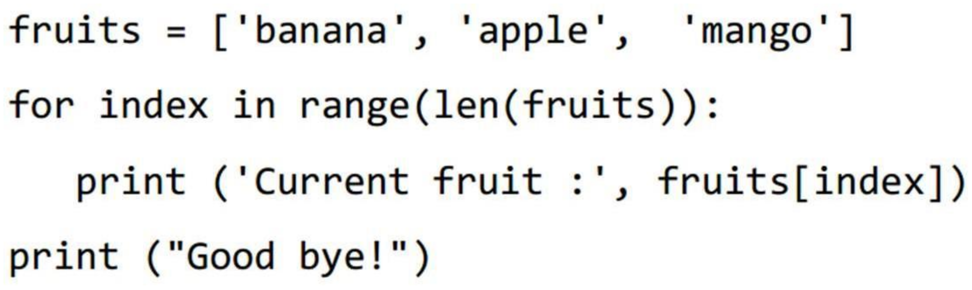


```
for var in list(range(5)):

 print (var)
```

Range



```
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # traversal of List sequence
   print ('Current fruit :', fruit)
```

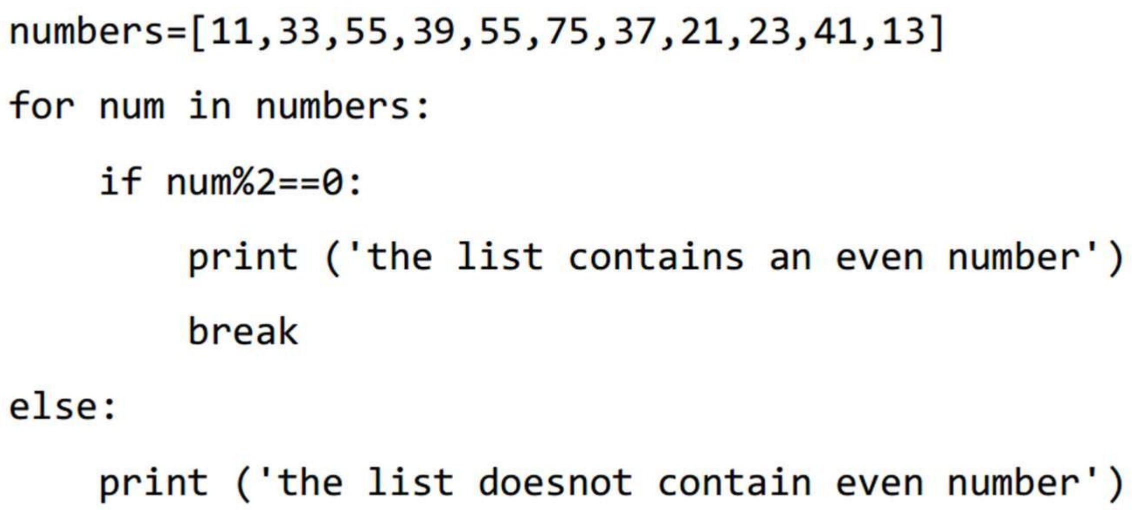


```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :', fruits[index])
print ("Good bye!")
```

Else in Loops

- Python supports having an **else** statement associated with a loop statement.
- If the **else** statement is used with a **for** loop, the **else** block is executed only if for loops terminates normally (and not by encountering break statement)
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false

Loops

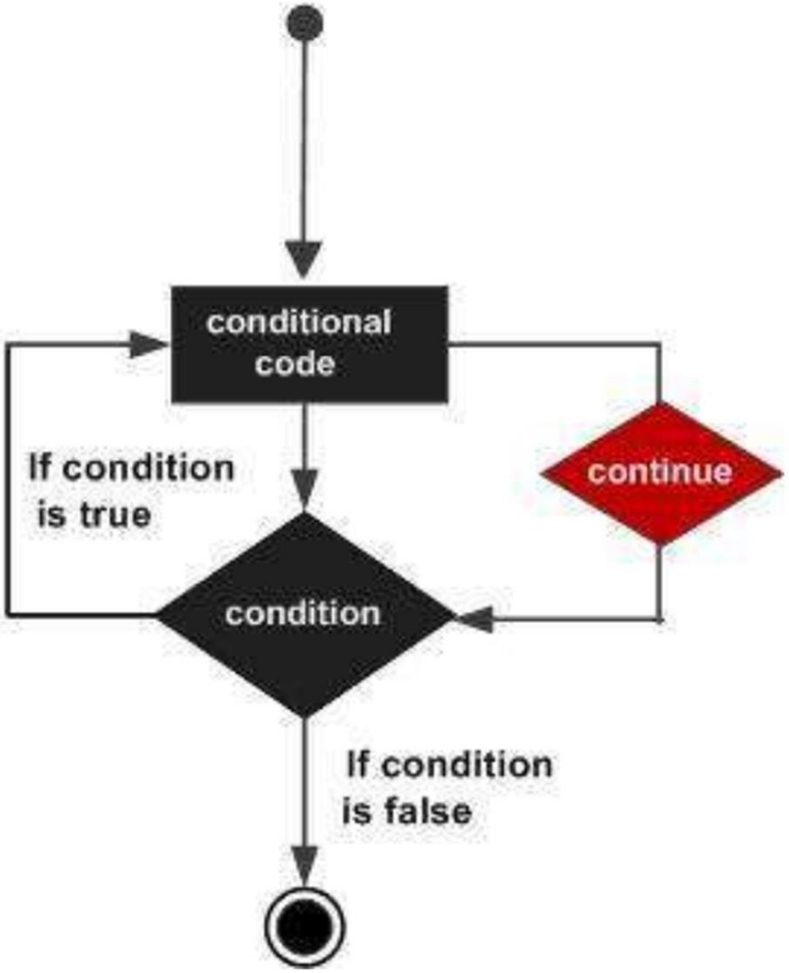


```
numbers=[11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:

    if num%2==0:

        print ('the list contains an even number')

        break

else:

    print ('the list doesnot contain even number')
```

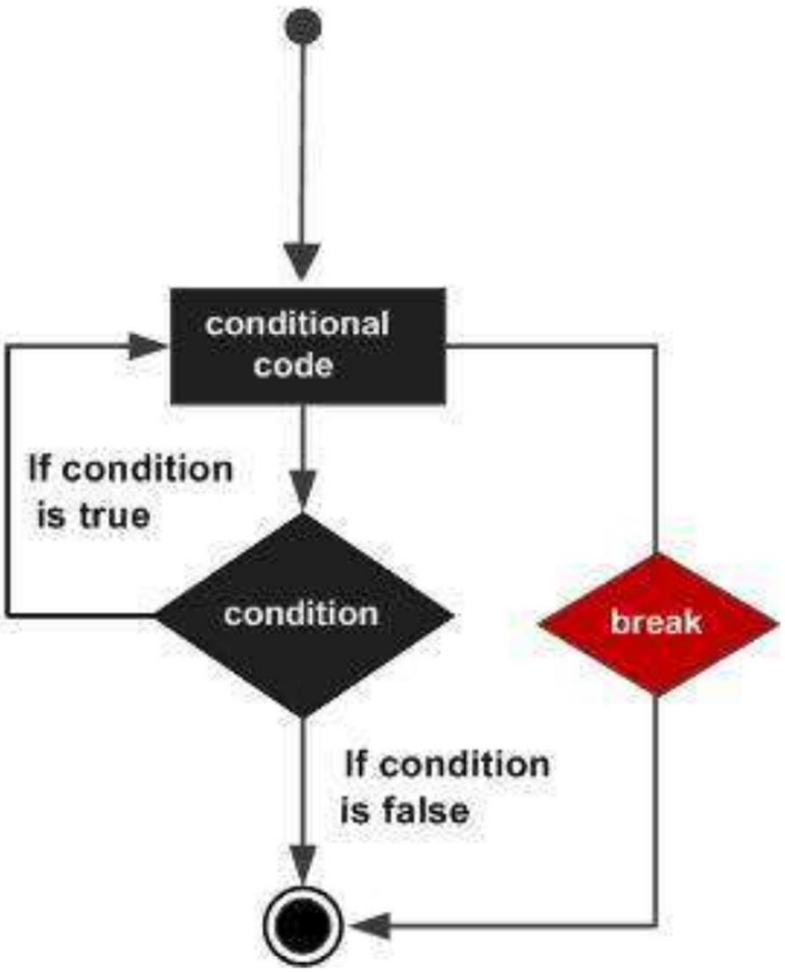
Loop Control Statements

- break
- continue
- pass
  - The **pass** statement is a *null* operation; nothing happens when it executes.
  - The **pass** statement is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs)

Continue

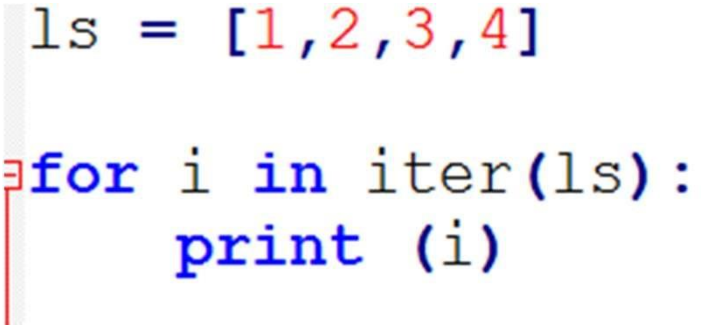

conditional code
If condition is true
continue
condition
If condition is false

Break

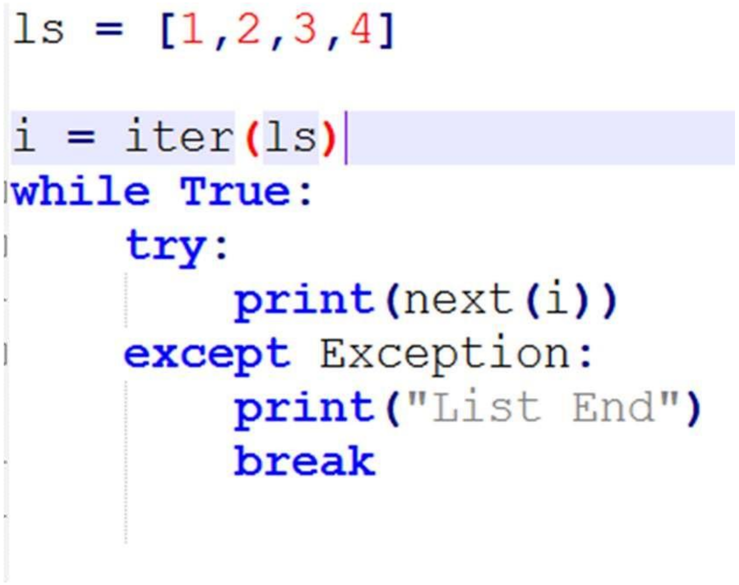

conditional code
If condition is true
condition
break
If condition is false

Iterator

- **Iterator** allows to traverse through all the elements of a collection
- implements two methods, **iter()** and **next()**
- String, List or Tuple objects can be used to create an Iterator



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

for i in iter(ls):
    print (i)
```

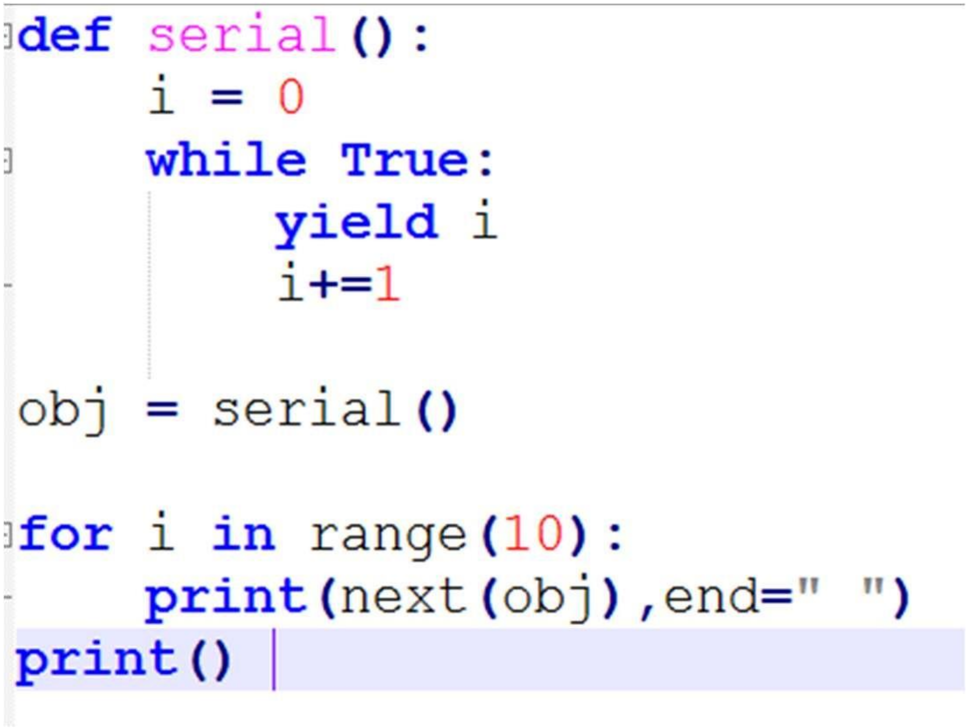


```
ls = [1,2,3,4]

i = iter(ls)
while True:
    try:
        print(next(i))
    except Exception:
        print("List End")
        break
```

Generator

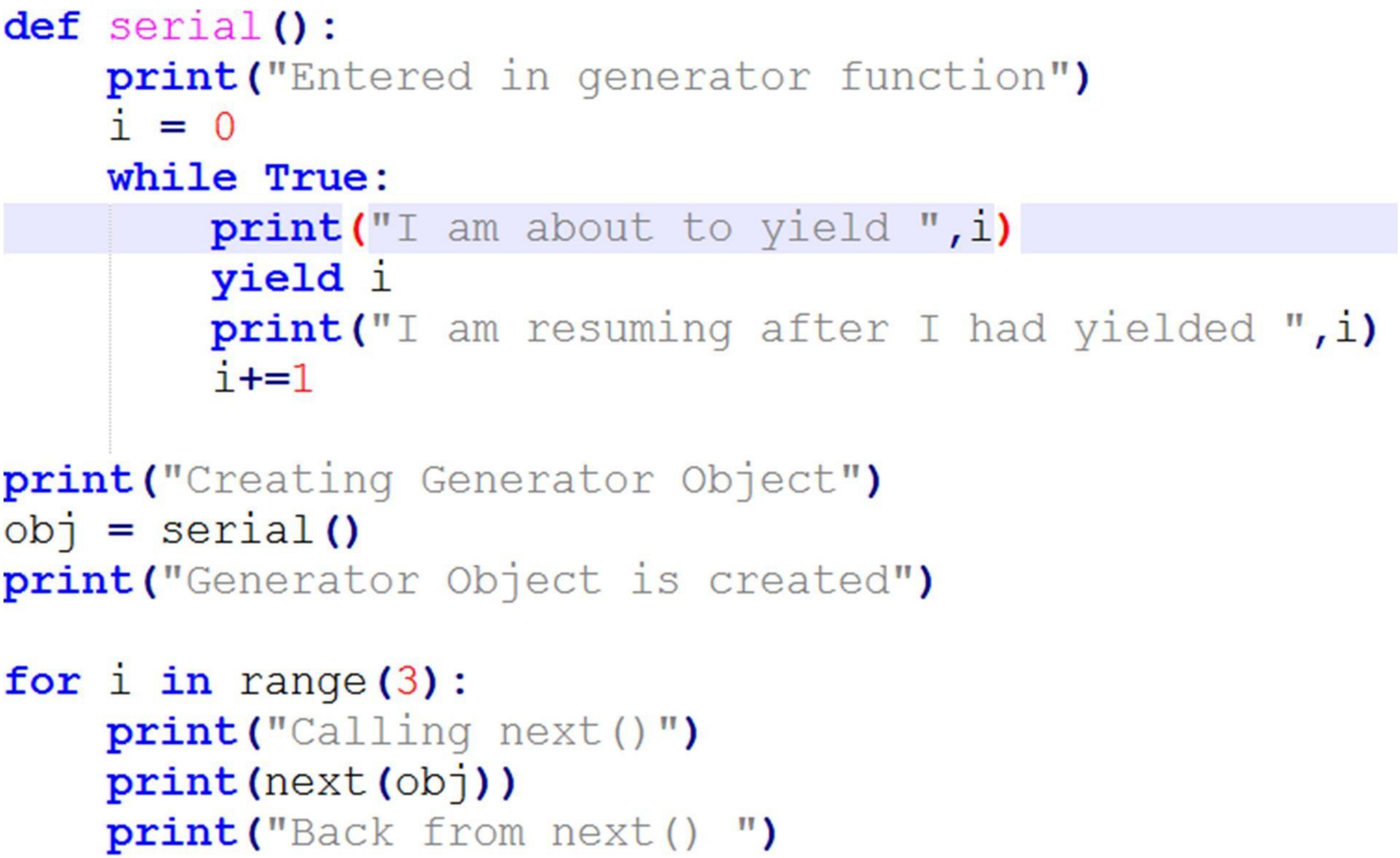
- A **generator** is a function that produces or yields a sequence of values using yield method.
- When a generator function is called, it returns a generator object without even beginning execution of the function.
- When the next() method is called for the first time, the function starts executing, until it reaches the yield statement, which returns the yielded value.
- The yield keeps track i.e. remembers the last execution and the second next() call continues from previous value.

Generator - Example



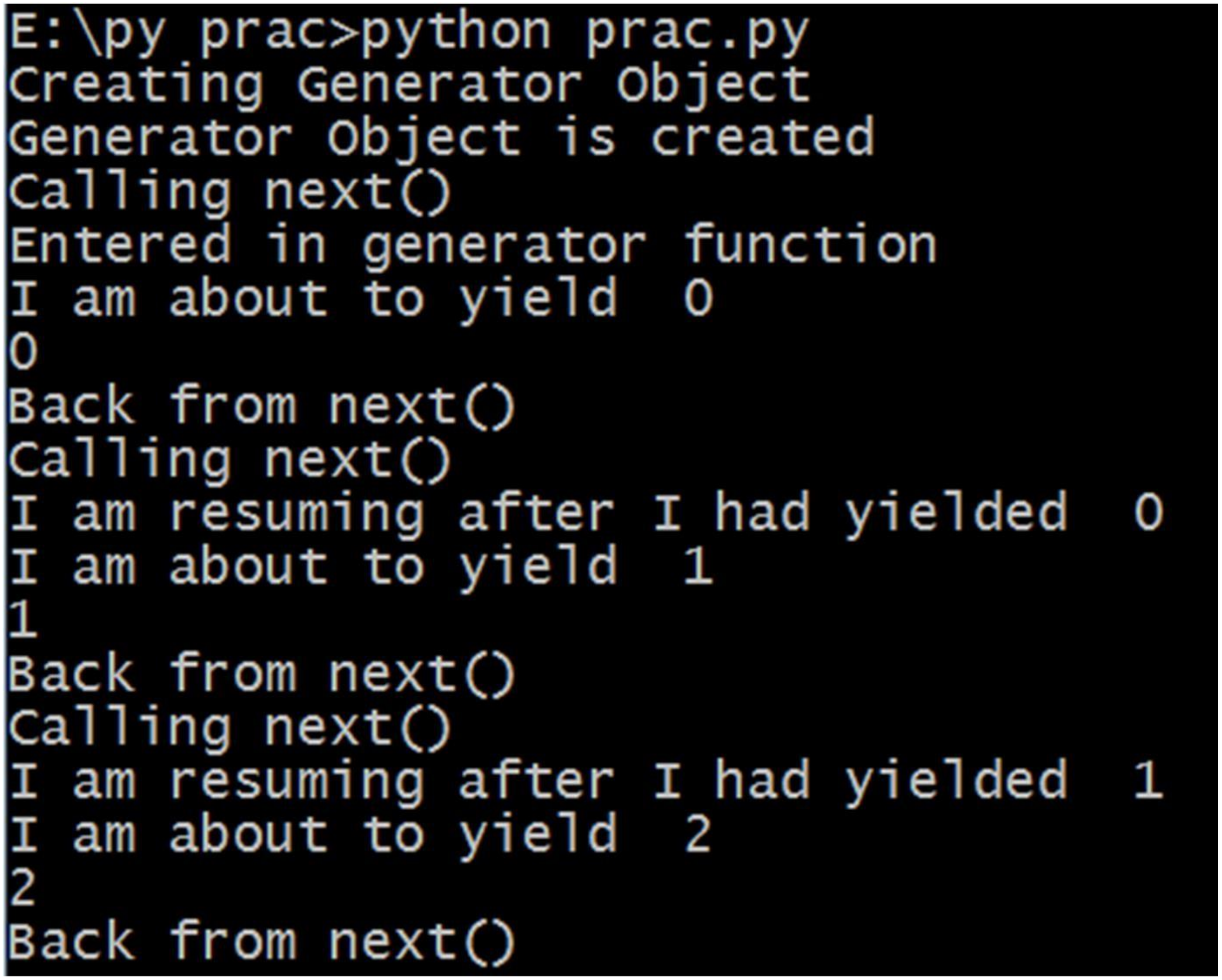
```python
def serial():
    i = 0
    while True:
        yield i
        i+=1

obj = serial()

for i in range(10):
    print(next(obj),end=" ")
print()
```

Generator – Example 2



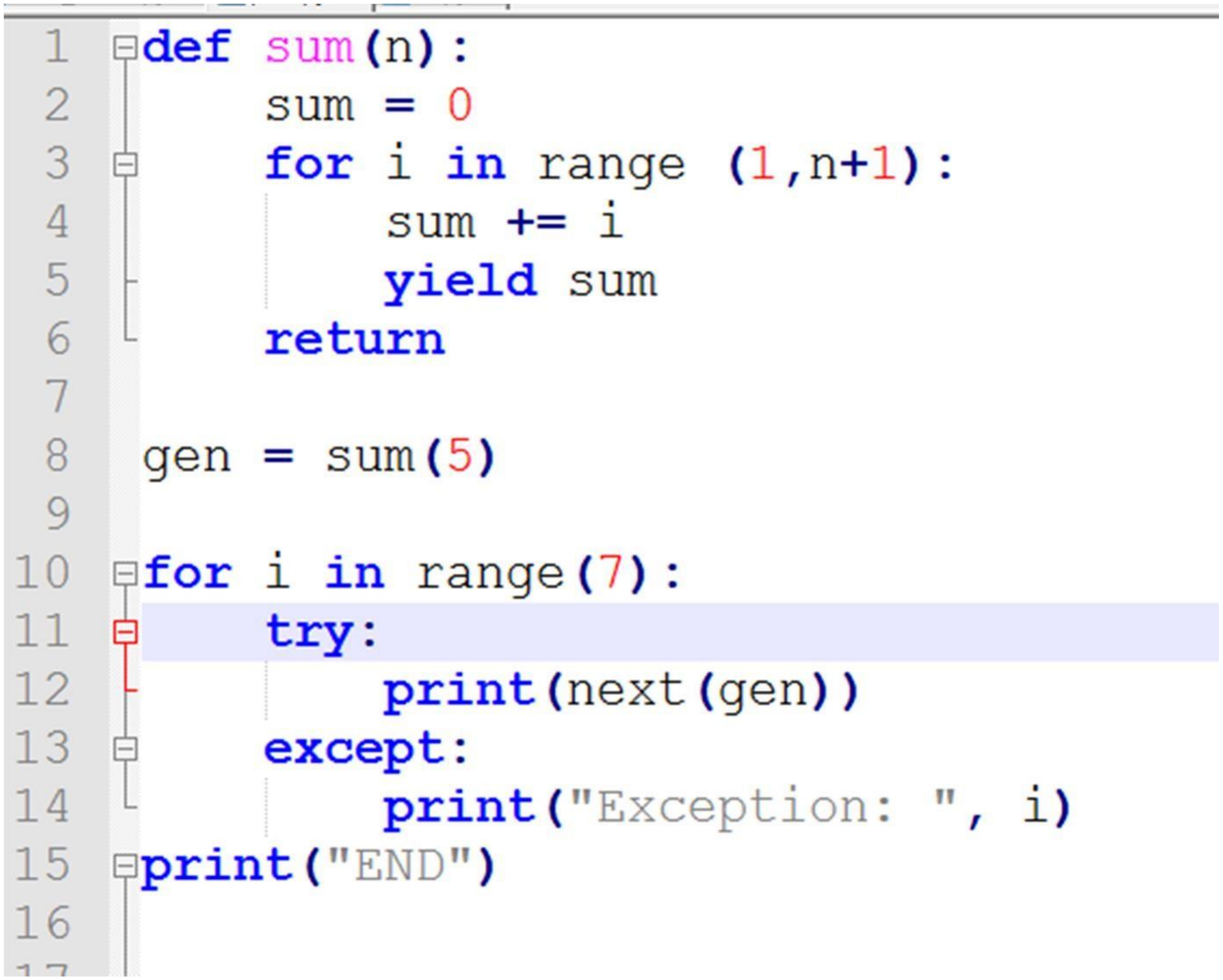
```python
def serial():
    print("Entered in generator function")
    i = 0
    while True:
        print("I am about to yield ",i)
        yield i
        print("I am resuming after I had yielded ",i)
        i+=1

print("Creating Generator Object")
obj = serial()
print("Generator Object is created")

for i in range(3):
    print("Calling next()")
    print(next(obj))
    print("Back from next()  ")
```

Example 2 Output



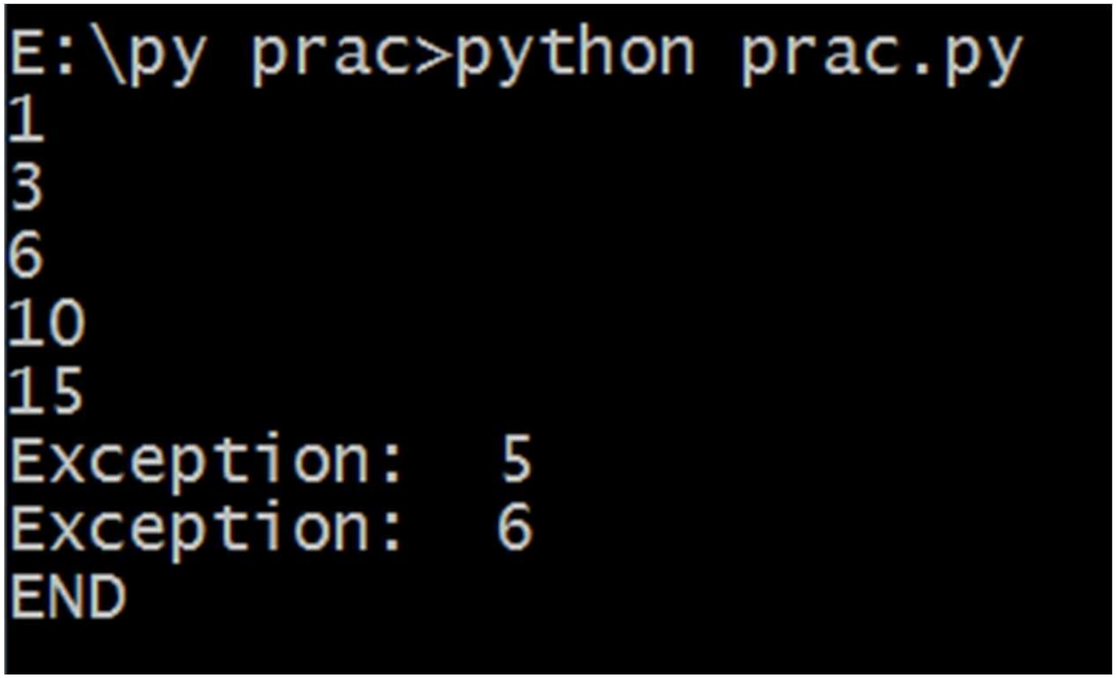
```
E:\py prac>python prac.py
Creating Generator Object
Generator Object is created
Calling next()
Entered in generator function
I am about to yield  0
0
Back from next()
Calling next()
I am resuming after I had yielded  0
I am about to yield  1
1
Back from next()
Calling next()
I am resuming after I had yielded  1
I am about to yield  2
2
Back from next()
```

Example 3



```python
def sum(n):
    sum = 0
    for i in range (1,n+1):
        sum += i
        yield sum
    return

gen = sum(5)

for i in range(7):
    try:
        print(next(gen))
    except:
        print("Exception: ", i)
print("END")
```

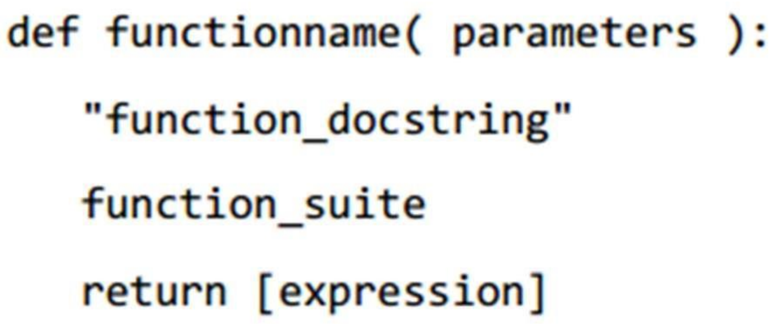
Output – Example 3



```
E:\py prac>python prac.py
1
3
6
10
15
Exception:  5
Exception:  6
END
```

FUNCTIONS

Structure



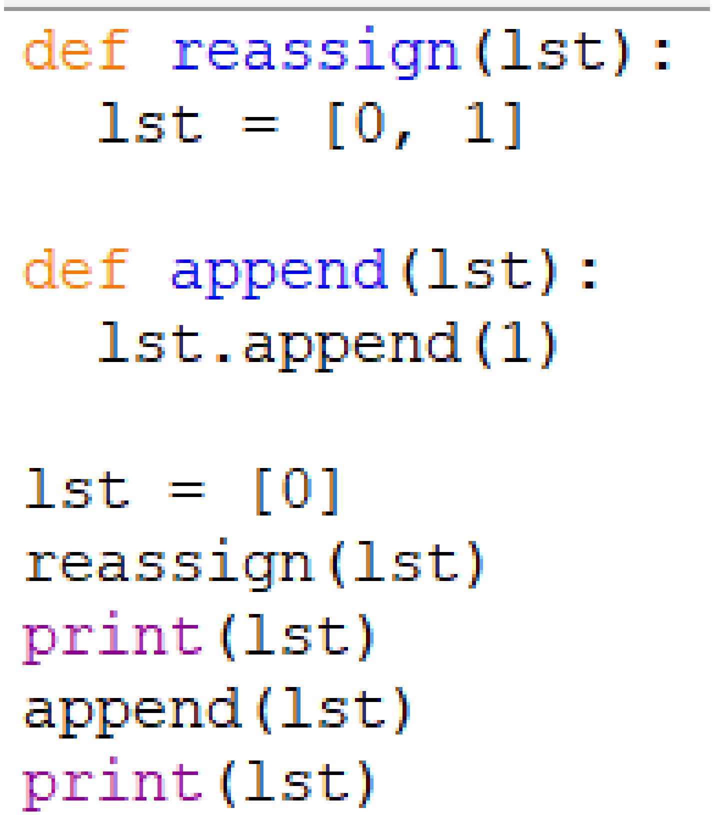
```
def functionname( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

- parameters can also be defined inside the parentheses
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- A return statement with no arguments is the same as return None
    - Can also be eliminated

Pass by Reference vs Pass by Value

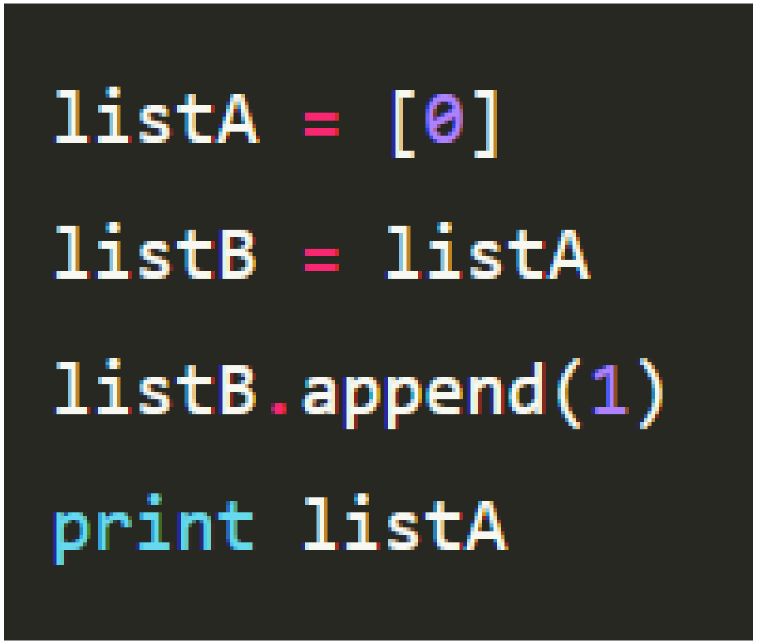
- All parameters (arguments) in the Python language are **passed-by-object-reference.**
  - Object references are passed by value
- Variable and the actual object are different.

Guess the Output



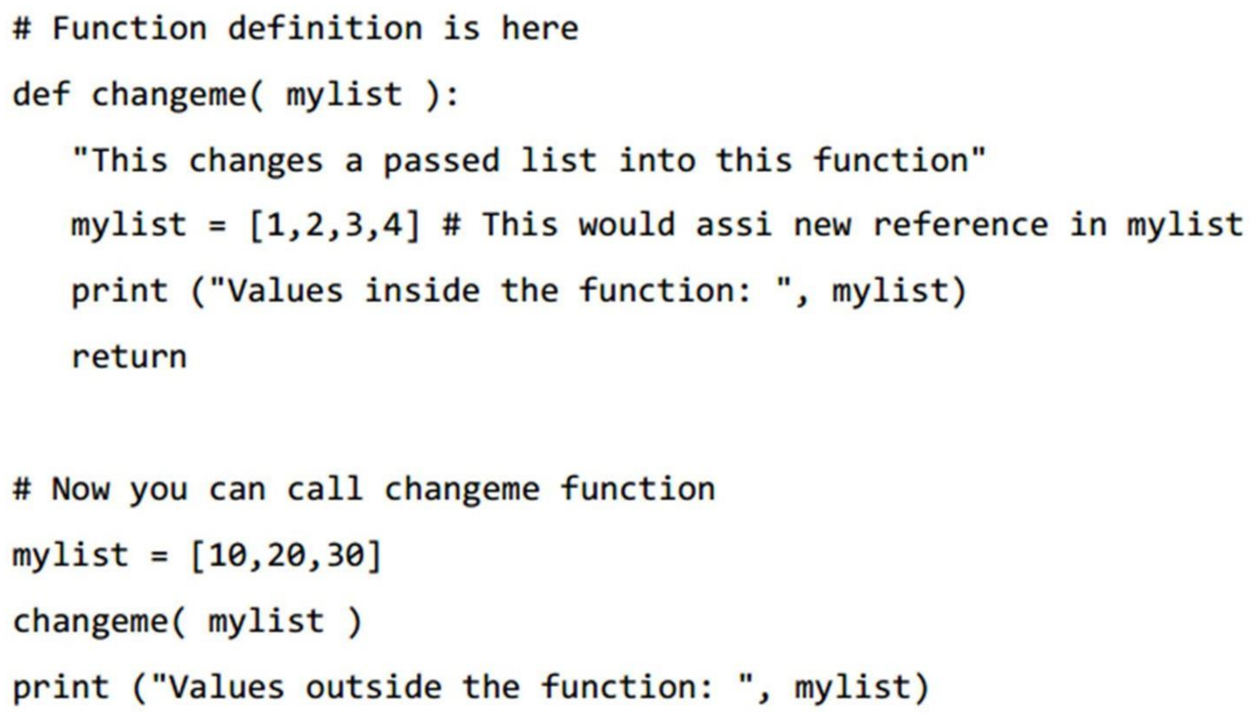
```python
def reassign(lst):
  lst = [0, 1]

def append(lst):
  lst.append(1)

lst = [0]
reassign(lst)
print(lst)
append(lst)
print(lst)
```

Guess the Output

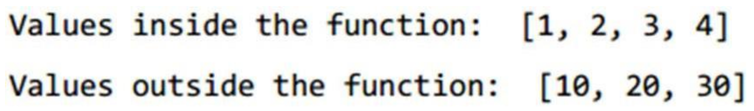


```
listA = [0]
listB = listA
listB.append(1)
print listA
```

Guess The Output



```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4] # This would assi new reference in mylist
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```



```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments-

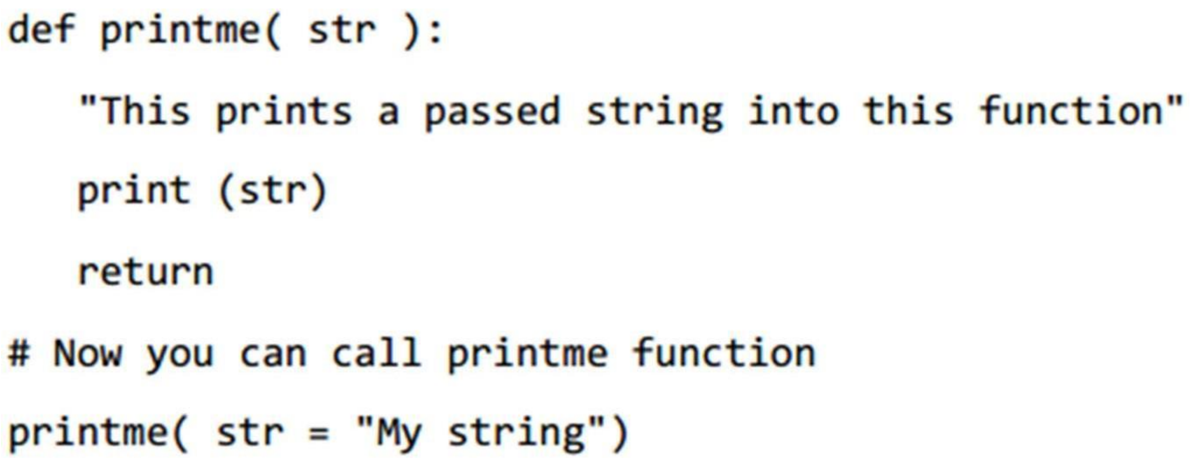
- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

- Required arguments are the arguments passed to a function in correct positional order.
  - typical parameters like C
- The number of arguments and their order in the function call should match exactly with the function definition.

Keyword Arguments

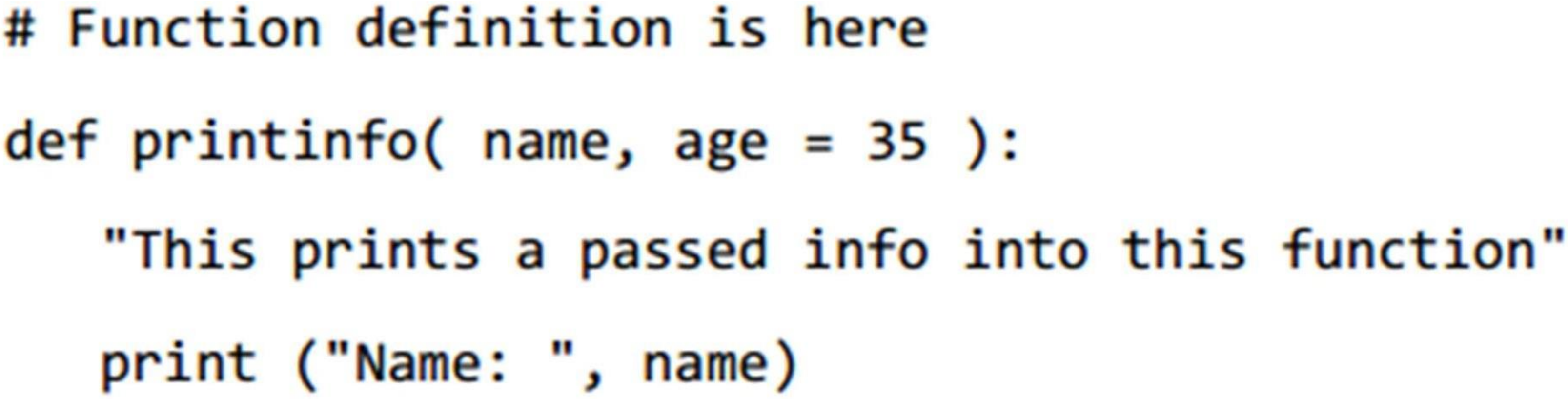
- Used to pass arguments by the parameter name.
- This allows to skip arguments or place them out of order



```
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return
# Now you can call printme function
printme( str = "My string")
```

```
# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Age ", age)
   return

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

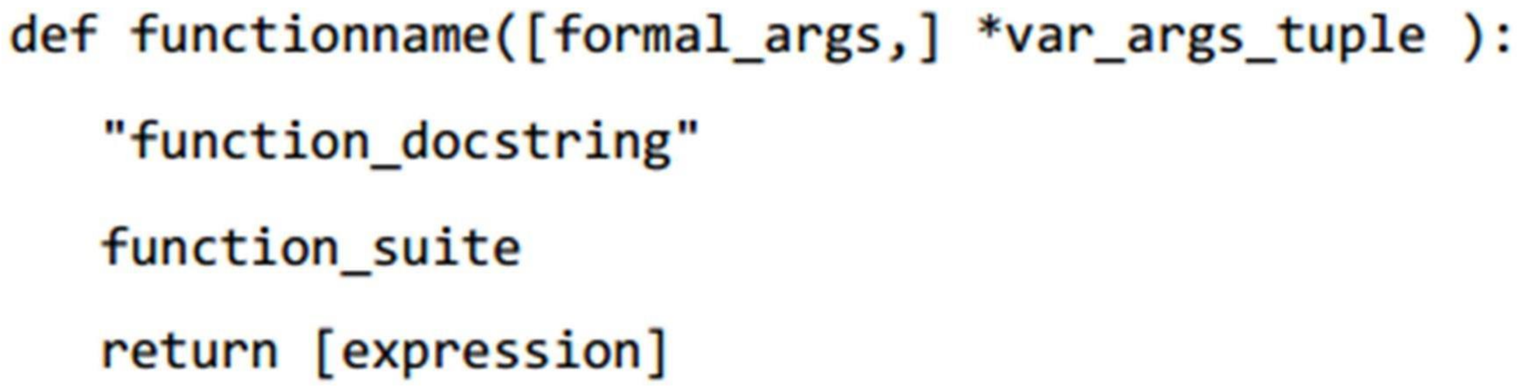
Default Arguments



```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print ("Name: ", name)
```

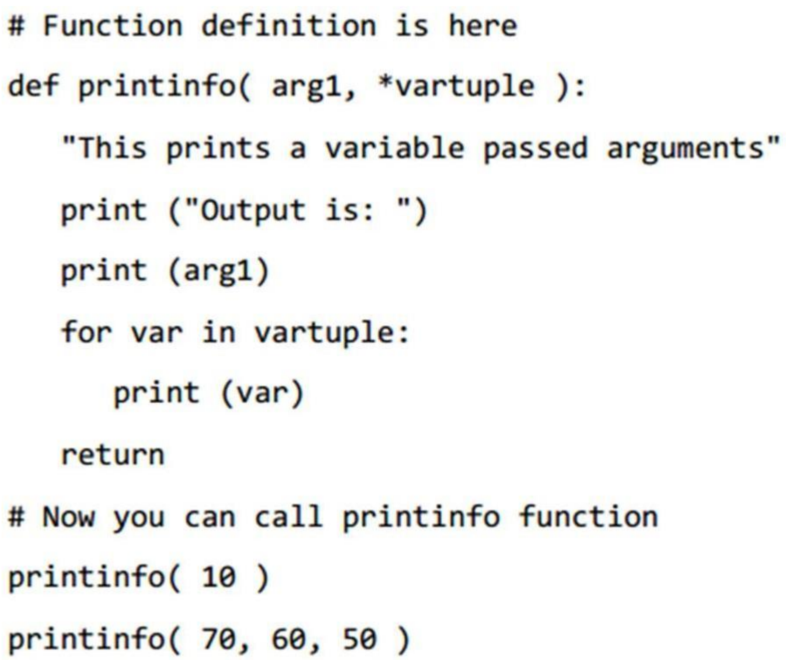
Variable-length Arguments

- *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

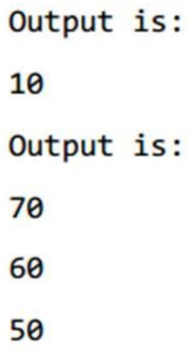


```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

Example

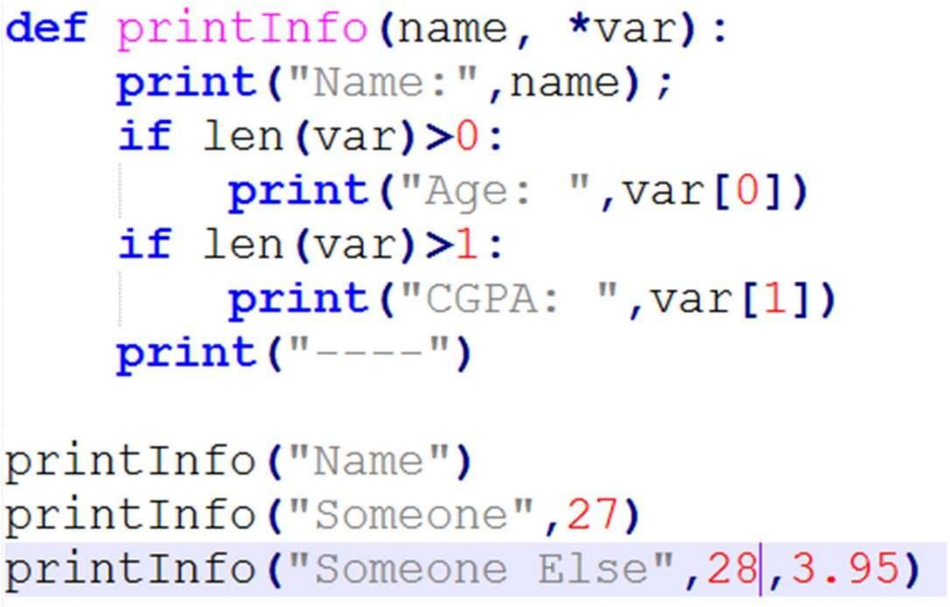


```
# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print ("Output is: ")
   print (arg1)
   for var in vartuple:
      print (var)
   return
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

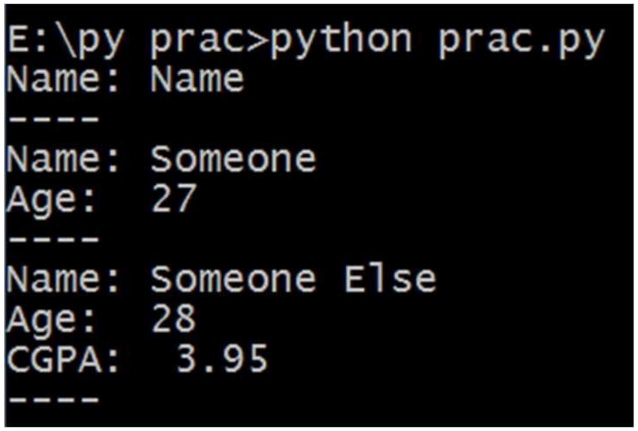


```
Output is:
10
Output is:
70
60
50
```

Example



```
def printInfo(name, *var):
    print("Name:",name);
    if len(var)>0:
        print("Age: ",var[0])
    if len(var)>1:
        print("CGPA: ",var[1])
    print("----")

printInfo("Name")
printInfo("Someone",27)
printInfo("Someone Else",28,3.95)
```



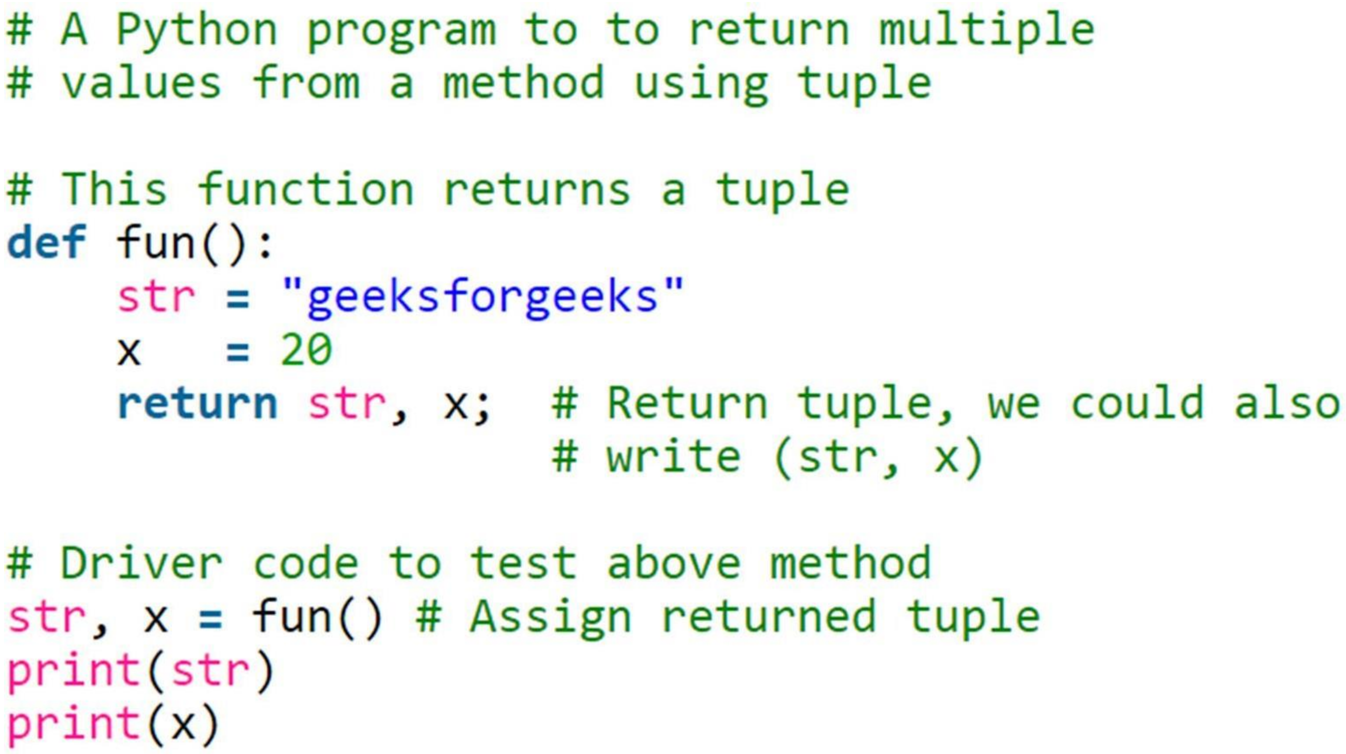
```
E:\py prac>python prac.py
Name: Name
----
Name: Someone
Age:  27
----
Name: Someone Else
Age:  28
CGPA:  3.95
----
```

Scope of Variables

- There are two basic scopes of variables in Python-
  - global variables
  - local variables
- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

Returning Multiple Values

- Can be done using class, tuples, list, or dictionary
- Most convenient by tuples



```python
# A Python program to to return multiple
# values from a method using tuple

# This function returns a tuple
def fun():
    str = "geeksforgeeks"
    x   = 20
    return str, x;  # Return tuple, we could also
                    # write (str, x)

# Driver code to test above method
str, x = fun() # Assign returned tuple
print(str)
print(x)
```