

PYTHON 3

BASIC SYNTAX

Python is an interpreted language

- You can write programs interactively using the interpreter
- You can also write scripts
 - File extension will be .py [eg. demo.py]
 - In console the script can be run by python command

Python Basic Concepts

- Identifier
- Reserved Words

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`
- Python does not have multiple-line commenting feature.
 - However, triple quotes can be utilized

User Input

- `input()`
 - takes the next line from console
- `input("\n\nPress the enter key to exit.")`
- `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)
 - All ints by default long
- 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 ”””

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', 70.2]  
abcd  
[786, 2.23]  
[2.23, 'john', 70.2]  
[123, 'john', 123, 'john']  
['abcd', 786, 2.23, 'john', 70.2, 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', 70.20000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.20000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.20000000000003, 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

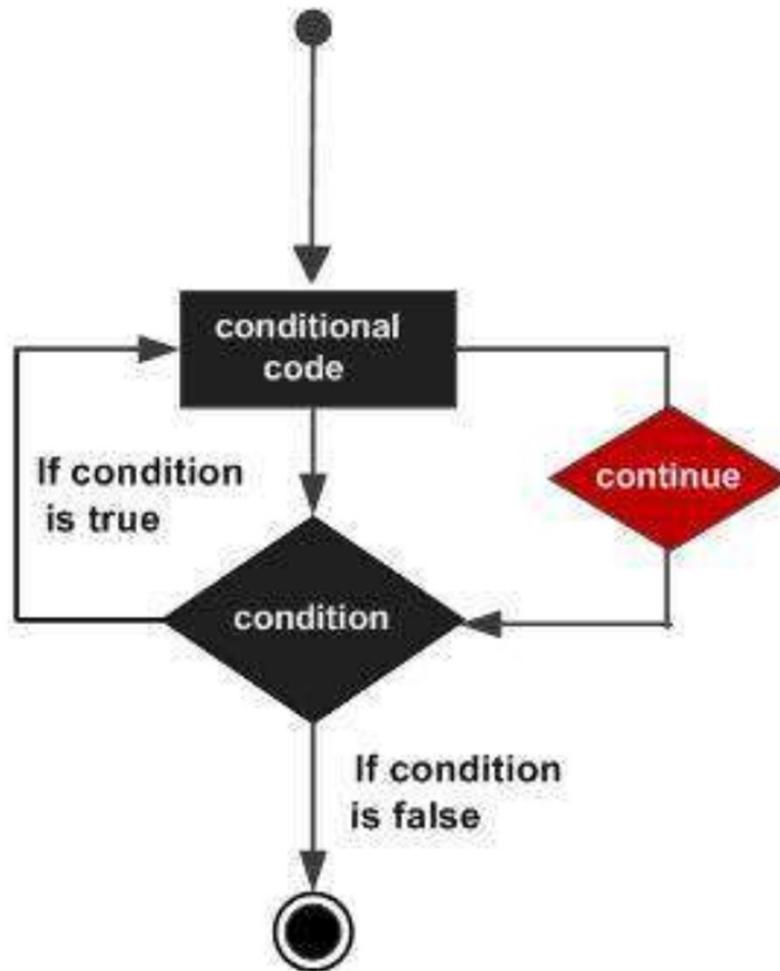
Loops

```
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')
```

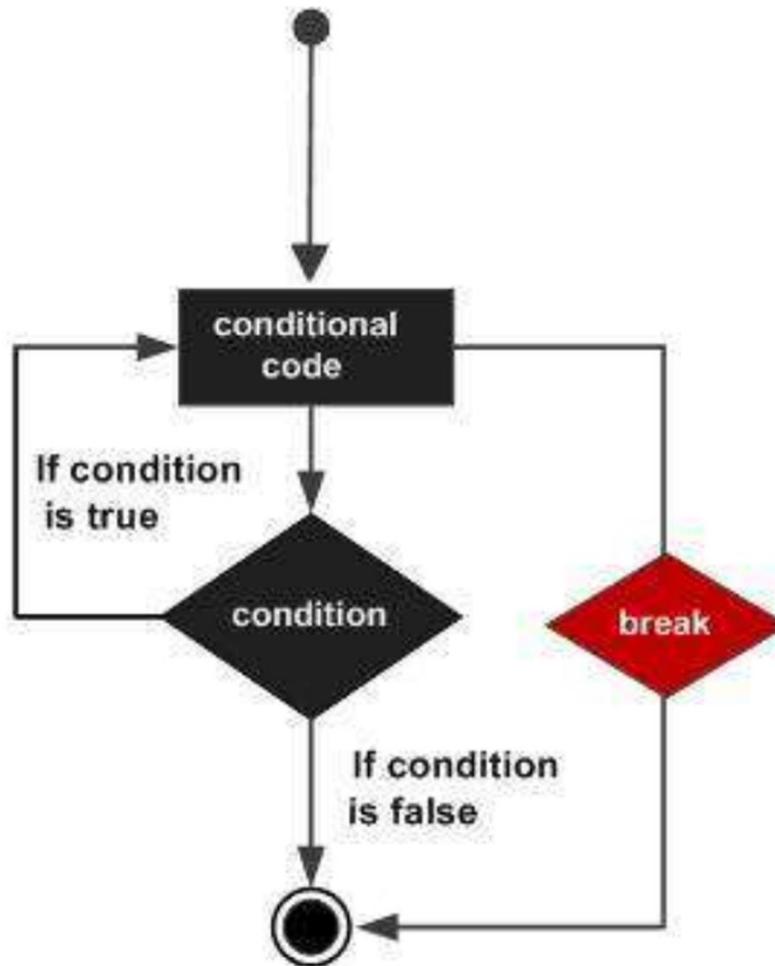
Loop Control Statements

- 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



Break



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

```
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

```
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

```
1 def sum(n):
2     sum = 0
3     for i in range(1, n+1):
4         sum += i
5         yield sum
6     return
7
8 gen = sum(5)
9
10 for i in range(7):
11     try:
12         print(next(gen))
13     except:
14         print("Exception: ", i)
15 print("END")
16
17
```

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

Pass byReference vsValue

- All parameters (arguments) in the Python language are passed by reference.
- It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

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)
```

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 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

```
# 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)
```

OBJECT ORIENTED PYTHON

Declaration

```
class ClassName:  
    'Optional class documentation string'  
    class_suite
```

The documentation string, can be accessed via *ClassName.__doc__*.

Example

```
class Employee:  
    'Common base class for all employees'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print "Total Employee %d" % Employee.empCount  
  
    def displayEmployee(self):  
        print ("Name : ", self.name, ", Salary: ", self.salary)
```

- The variable *empCount* is a class variable whose value is shared among all the instances of a in this class. (like a static member of Java)
- This can be accessed as *Employee.empCount* from inside the class or outside the class

Example

```
class Employee:  
    'Common base class for all employees'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print "Total Employee %d" % Employee.empCount  
  
    def displayEmployee(self):  
        print ("Name : ", self.name, ", Salary: ", self.salary)
```

- The first method `__init__()` is the initialization method
 - Not same as constructors
 - it is called when a new instance of this class is created.

Example

```
class Employee:  
    'Common base class for all employees'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print "Total Employee %d" % Employee.empCount  
  
    def displayEmployee(self):  
        print ("Name : ", self.name, ", Salary: ", self.salary)
```

- first argument to each method is *self*.
- While calling a method from a class, Python adds the *self* argument automatically

Creating Instance Objects

This would create first object of Employee class

```
emp1 = Employee("Zara", 2000)
```

This would create second object of Employee class

```
emp2 = Employee("Manni", 5000)
```

```
    emp1.displayEmployee()
```

```
    emp2.displayEmployee()
```

```
    print ("Total Employee %d" % Employee.empCount)
```

```
emp1.salary = 7000
```

```
emp1.name = 'xyz' # Modify 'age' attribute.
```

```
del emp1.salary # Delete 'age' attribute.
```

Setting and getting attributes

Instead of using the normal statements to access attributes, you can use the following functions-

- The **getattr(obj, name[, default])**: to access the attribute of object.
- The **hasattr(obj, name)**: to check if an attribute exists or not.
- The **setattr(obj, name, value)**: to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)**: to delete an attribute.

```
hasattr(emp1, 'salary')      # Returns true if 'salary' attribute exists
getattr(emp1, 'salary')      # Returns value of 'salary' attribute
setattr(emp1, 'salary', 7000) # Set attribute 'age' at 8
delattr(emp1, 'salary')      # Delete attribute 'age'
```

Class variable vs. Instance Variable

```
class MyEmp:  
    salary = 3000;  
  
    def __init__(self, salary):  
        self.salary = salary  
  
    def prSlr(self):  
        print(self.salary, " ", MyEmp.salary)  
  
em = MyEmp("40000")  
em.prSlr()
```

```
40000 3000
```

Examples – You can Change Type

```
class MyEmp:  
    salary = 3000;  
  
    def __init__(self, salary):  
        self.salary = salary  
  
    def prSlr(self):  
        print(self.salary, " ", MyEmp.salary)  
  
  
em = MyEmp("40000")  
em.prSlr()  
print(type(em.salary))  
em.salary = 50000 #type changed  
em.prSlr()  
print(type(em.salary))
```

40000 3000
<class 'str'>
50000 3000
<class 'int'>

You can even create a new member

```
| class MyEmp:  
|     salary = 3000;  
  
|     def __init__(self, salary):  
|         self.salary = salary  
  
|     def prSlr(self):  
|         print(self.salary, " ", MyEmp.salary)  
  
  
em = MyEmp(5000);  
em.name = "Kuddus" # you can create  
# new member variable  
print(em.name)
```

You can even create a new member

```
class MyEmp:  
    salary = 3000;  
  
    def __init__(self, salary):  
        self.salary = salary  
  
    def prSlr(self):  
        print(self.salary, " ", MyEmp.salary)  
  
em = MyEmp(5000);  
print(hasattr(em, "name"))  
em.name = "Kuddus" # you can create  
                    # new member variable  
print(hasattr(em, "name"))  
print(em.name)
```

False
True
Kuddus

- Adding attribute in an object doesn't change the class

```
em = MyEmp(5000);
print(hasattr(em, "name"))
em.name = "Kuddus" # you can create
                  # new member variable
print(hasattr(em, "name"))
print(em.name)

em2 = MyEmp(2000)
print(hasattr(em2, "name")) # False

em2 = em
print(hasattr(em2, "name")) # True
```

These two codes are exactly equivalent

```
class Foo(object):
    def __init__(self, bar):
        self.bar = bar

foo = Foo(5)
```

And this code:

```
class Foo(object):
    pass

foo = Foo()
foo.bar = 5
```

Objects and members

- The members that a class will have is not restricted in the time of class declaration
 - In fact, there is no notion of declaration !!
 - However, there is way to restrict this property
 - Some built in types also do not allow this
- **Most instances store their attributes in a dictionary**
 - a regular Python dictionary like you'd define with {}
- Check this SO answer for details:
 - <https://stackoverflow.com/questions/12569018/why-is-adding-attributes-to-an-already-instantiated-object-allowed>

Built-In Class Attributes

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute –

- **`__dict__`**: Dictionary containing the class's namespace.
 - **`__doc__`**: Class documentation string or none, if undefined.
 - **`__name__`**: Class name.
 - **`__module__`**: Module name in which the class is defined. This attribute is "`__main__`" in interactive mode.
- `__bases__`**: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

obj.__dict__

```
In[4]: class Dummy:  
....:     pass  
....:  
In[5]: d = Dummy()  
In[6]: d.__dict__  
Out[6]:  
{}  
In[7]: d.a = 1  
In[8]: d.__dict__  
Out[8]:  
{'a': 1}  
In[9]: d.b = "B"  
In[10]: d.__dict__  
Out[10]:  
{'a': 1, 'b': 'B'}  
In[11]: d.c = "You cant see me"  
In[12]: d.__dict__  
Out[12]:  
{'a': 1, 'b': 'B', 'c': 'You cant see me'}
```

obj.__dict__

```
In[14]: def meth():
...:     print("Add Me")
...:
In[15]: def block():
...:     print("Add Meh")
...:
In[16]: d.block = block
In[17]: d.block()
Add Meh
In[18]: d.__dict__
Out[18]:
{'a': 1, 'b': 'B', 'block': <function __main__.block>, 'c': 'You cant see me'}
```

Garbage Collection

- When a object is no longer referenced, Python automatically destroys it and reclaims the memory

Destructor Function

- `__del__()`
- The destruction function is called when an object is going to be destroyed

Example

```
class MyEmp:

    def __init__(self, name):
        self.name = name

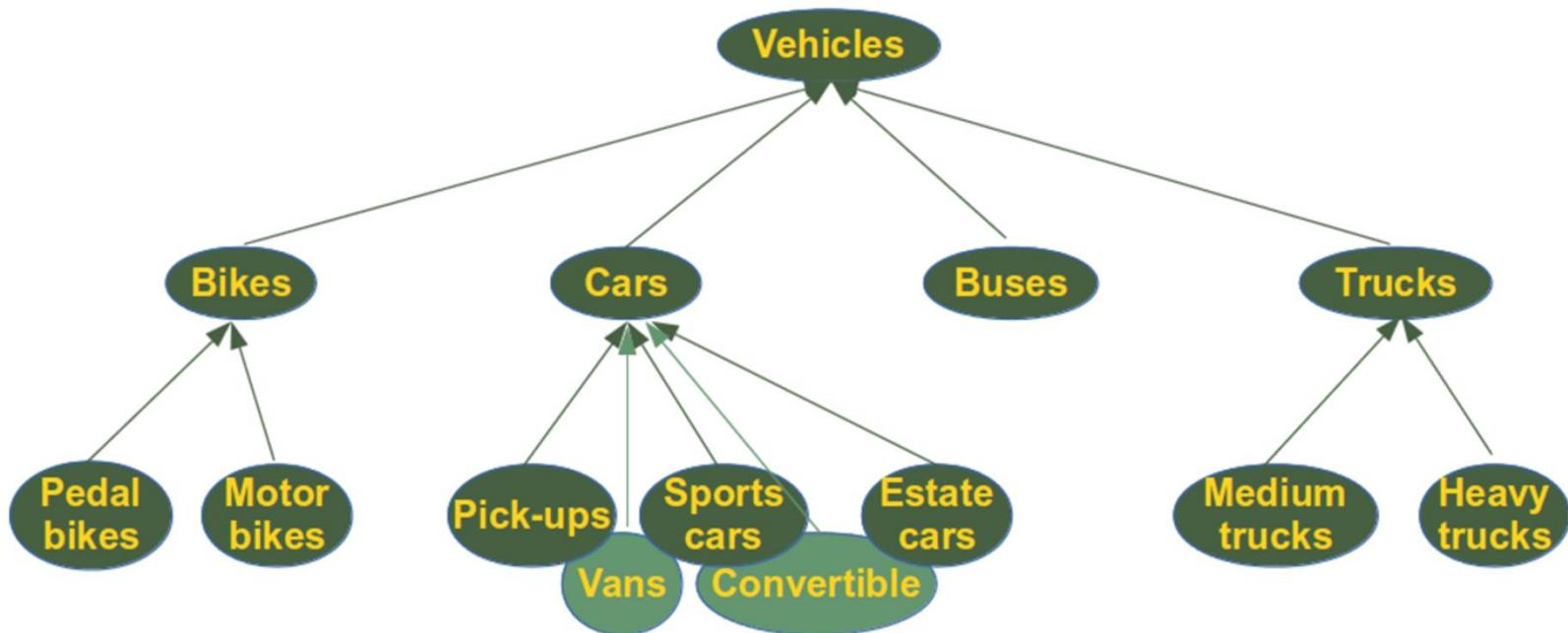
    def __del__(self):
        print(self.name, " is going to be destroyed!!")

emp1 = MyEmp("EMP1")
print(emp1.name)
emp1 = MyEmp("EMP2")
print(emp1.name)
```

```
| Running F:/PyCharmWS/Prac/dummy.py
| EMP1
| EMP1  is going to be destroyed!!
| EMP2
```

INHERITANCE

Inheritance



Inheritance

- You create a class by deriving it from a pre-existing class
- The child class inherits the attributes of its parent class
 - you can use those attributes as if they were defined in the child class.
- A child class can also override data members and methods from the parent

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    'Optional class documentation string'  
    class_suite
```

Example

```
class Polygon:

    def __init__(self):
        print("In Polygon-Constructor")

    def intro(self):
        print("I am a polygon")

class Triangle(Polygon):
    def __init__(self):
        print("In Triangle-Constructor")

    def myIntro(self):
        print("I am a triangle")

t = Triangle()
t.intro()
t.myIntro()
```

```
Running F:/PyCharmWS/Prac/dummy.py
In Triangle-Constructor
I am a polygon
I am a triangle
```

```
)class Polygon:  
)  
    def __init__(self):  
        print("In Polygon-Constructor")  
    self.m = 0  
  
    def intro(self):  
        print("I am a polygon")  
  
class Triangle(Polygon):  
)  
    def __init__(self):  
        super().__init__()  
        print("In Triangle-Constructor")  
  
    def myIntro(self):  
        print("I am a triangle")  
  
t = Triangle()  
t.intro()  
t.myIntro()  
print(t.m) #Attribute error
```

```
class Polygon:

    def __init__(self):
        print("In Polygon-Constructor")
        self.m = 0

    def intro(self):
        print("I am a polygon")


class Triangle(Polygon):
    def __init__(self):
        super().__init__()
        print("In Triangle-Constructor")

    def myIntro(self):
        print("I am a triangle")

t = Triangle()
t.intro()
t.myIntro()
print(t.m) #prints 0 now
```

Overriding

```
)class Polygon:

)    def __init__(self):
)        print("In Polygon-Constructor")
)        self.m = 0

)    def intro(self):
)        print("I am a polygon")

)class Triangle(Polygon):
)    def __init__(self):
)        super().__init__()
)        print("In Triangle-Constructor")

)    def intro(self):
)        print("I am a triangle")

t = Triangle()
t.intro() # prints triangle
```

Inheriting Multiple Class

```
class A:      # define your class A  
.....  
class B:      # define your calss B  
.....  
class C(A, B):  # subclass of A and B  
.....
```

You can use `issubclass()` or `isinstance()` functions to check a relationship of two classes and instances.

- The **`issubclass(sub, sup)`** boolean function returns True, if the given subclass **`sub`** is indeed a subclass of the superclass **`sup`**.
- The **`isinstance(obj, Class)`** boolean function returns True, if *obj* is an instance of class *Class* or is an instance of a subclass of *Class*.

MORE ON NUMBERS

Numbers

- They are immutable data types.
- This means, changing the value of a number data type results in a newly allocated object

Number objects are created when you assign a value to them. For example-

```
var1 = 1  
var2 = 10
```

Deleting reference

You can also delete the reference to a number object by using the **del** statement. The syntax of the **del** statement is –

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the **del** statement. For example-

```
del var  
del var_a, var_b
```

Deleting reference

```
i = 5
print(i)
del i
print(i)
```

```
E:\py prac>python prac.py
5
Traceback (most recent call last):
  File "prac.py", line 4, in <module>
    print(i)
NameError: name 'i' is not defined
```

int (signed integers)

- positive or negative whole numbers
- Integers in Python 3 are of unlimited size.
 - Python 2 has two integer types - int and long.
There is no '**long integer**' in Python 3 anymore.
- It is possible to represent an integer in hexa-decimal or octal form.

```
>>> number = 0xA0F #Hexa-decimal  
  
>>> number  
  
2575
```

```
>>> number=0o37 #Octal  
  
>>> number
```

float (floating point real values)

- represent real numbers
- Floats may also be in scientific notation, with E or e indicating the power of 10
 - $2.5\text{e}2 = 2.5 \times 10^2 = 250$

complex (complex numbers)

- are of the form $a + bj$
 - where a and b are floats
 - j (or J) represents the square root of -1

Number Type Conversion

- Python converts numbers internally in an expression containing mixed types to a common type for evaluation.

Explicit Conversion

Type **int(x)** to convert x to a plain integer.

Type **long(x)** to convert x to a long integer.

Type **float(x)** to convert x to a floating-point number.

Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.

Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions.

Some Useful Functions

- `abs(x), fabs(x)`
- `ceil(x), floor(x)`
- `cmp(x,y)`
- `exp(x)`
- `log(x), log10(x)`
- `sqrt(x)`
- `max(x1,x2,...)`
- `min(x1,x2,...)`

abs(x), fabs(x)

The **fabs()** method returns the absolute value of x. Although similar to the abs() function, there are differences between the two functions. They are-

- abs() is a built in function whereas fabs() is defined in math module.
- fabs() function works only on float and integer whereas abs() works with complex number also.

Some Useful Functions

- **modf(x)**
 - The fractional and integer parts of x in a two-item tuple.
 - Both parts have the same sign as x.
 - The integer part is returned as a float.

modf(x)

```
math.modf(100.12) : (0.1200000000000455, 100.0)
math.modf(100.72) : (0.7199999999999989, 100.0)
math.modf(119) : (0.0, 119.0)
math.modf(math.pi) : (0.14159265358979312, 3.0)
```

Some Useful Functions

- **round(x [,n])**
 - x rounded to n digits from the decimal point.
 - Python rounds away from zero as a tie-breaker:
 - `round(0.5)` is 1.0 and `round(-0.5)` is -1.0

Some Useful Functions

round(70.23456) : 70

round(56.659,1) : 56.7

round(80.264, 2) : 80.26

round(100.000056, 3) : 100.0

round(-100.000056, 3) : -100.0

Random Number Functions

```
import random  
random.random()
```

Function	Description
choice(seq)	A random item from a list, tuple, or string.
randrange ([start,] stop [,step])	A randomly selected element from range(start, stop, step).
random()	A random float r, such that 0 is less than or equal to r and r is less than 1.

Random Number Functions

seed([x])	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
shuffle(lst)	Randomizes the items of a list in place. Returns None.
uniform(x, y)	A random float r, such that x is less than or equal to r and r is less than y.

Trigonometric Functions

acos(x)	
asin(x)	
atan(x)	
hypot(x, y)	Return the Euclidean norm, $\sqrt{x*x + y*y}$.
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

```
import math
```

Constants
math.pi
math.e

MORE ON STRINGS

Strings are immutable

```
In[1]: str = "Hello"
In[2]: str[1] = 'B'
Traceback (most recent call last):
  File "C:\Program Files\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py", line 2881, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-2-f8d39658be9a>", line 1, in <module>
    str[1] = 'B'
TypeError: 'str' object does not support item assignment
```

In Python3 all strings are Unicode

```
In[12]: print("আমার সোনার বাংলা")
আমার সোনার বাংলা
```

What happens here ?

```
a = "Dog"
b = "eats"
c = "treats"
|
print ~~(a)
a = a + " " + b + " " + c
print ~~(a)
```

String Formatting

- Like printf
- print(string % tuple)

```
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

When the above code is executed, it produces the following result –

```
My name is Zara and weight is 21 kg!
```

There are many String functions

- Find it from the book and play around

MORE ON LISTS

Lists are Sequence

- The most basic data structure in Python is the **sequence**
 - Each element has an index
- Lists, Tuples, Strings are sequences
- There are certain things you can do with all the sequence types.
 - These operations include indexing, slicing, adding, multiplying, and checking for membership

Negative index is possible

- Try it yourself

More List Functions

len(list)

Gives the total length of the list.

max(list)

Returns item from the list with max value.

min(list)

Returns item from the list with min value.

list(seq)

Converts a tuple into list.

More List Functions

list.append(obj)

Appends object obj to list

list.count(obj)

Returns count of how many times obj occurs in list

list.extend(seq)

Appends the contents of seq to list

list.index(obj)

Returns the lowest index in list that obj appears

list.insert(index, obj)

Inserts object obj into list at offset index

GuessTheOutput

```
In [2]: l = [1, 2, 3, 4]
In [3]: l2 = [5, 6]
In [4]: l.append(l2)
```

GuessTheOutput

```
In[2]: l = [1,2,3,4]
In[3]: l2 = [5,6]
In[4]: l.append(l2)
In[5]: l
Out[5]:
[1, 2, 3, 4, [5, 6]]
```

More List Functions

list.pop(obj=list[-1])

Removes and returns last object or obj from list

list.remove(obj)

Removes object obj from list

list.reverse()

Reverses objects of list in place

list.sort([func])

Sorts objects of list, use compare func if given

Deleting a value from list

```
#!/usr/bin/python3

list = ['physics', 'chemistry', 1997, 2000]

print (list)

del list[2]

print ("After deleting value at index 2 : ", list)
```

```
['physics', 'chemistry', 1997, 2000]
```

```
After deleting value at index 2 :  ['physics', 'chemistry', 2000]
```

List Comprehension

```
string = 'Hello'  
L = [1,14,5,9,12]  
M = ['one', 'two', 'three', 'four', 'five', 'six']
```

List comprehension	Resulting list
[0 for i in range (10)]	[0,0,0,0,0,0,0,0,0,0]
[i**2 for i in range (1,8)]	[1,4,9,16,25,36,49]
[i*10 for i in L]	[10,140,50,90,120]
[c*2 for c in string]	['HH','ee','ll','ll','oo']
[m[0] for m in M]	['o','t','t','f','f','s']
[i for i in L if i<10]	[1,5,9]
[m[0] for m in M if len(m)==3]	['o','t','s']

Practice

- **Example 1** Write a program that generates a list L of 50 random numbers between 1 and 100.
- **Example 2** Replace each element in a list L with its square
- **Example 3** Count how many items in a list L are greater than 50
- **Example 4** Given a list L that contains numbers between 1 and 100, create a new list whose first element is how many ones are in L, whose second element is how many twos are in L, etc.

Example 1 Write a program that generates a list `L` of 50 random numbers between 1 and 100.

```
L = [randint(1,100) for i in range(50)]
```

Example 2 Replace each element in a list `L` with its square.

```
L = [i**2 for i in L]
```

Example 3 Count how many items in a list `L` are greater than 50.

```
len([i for i in L if i>50])
```

Example 4 Given a list `L` that contains numbers between 1 and 100, create a new list whose first element is how many ones are in `L`, whose second element is how many twos are in `L`, etc.

```
frequencies = [L.count(i) for i in range(1,101)]
```

Another example The `join` method can often be used with list comprehensions to quickly build up a string. Here we create a string that contains a random assortment of 1000 letters.

```
from random import choice
alphabet = 'abcdefghijklmnopqrstuvwxyz'
s = ''.join([choice(alphabet) for i in range(1000)])
```

MORE ON DICTIONARIES

Dictionary – Restrictions on KEY

- Only one item per key is allowed
- Value can be any type of object
- However, key must be IMMUTABLE
- For using a user defined object as a key you need to override two methods
 - `__hash__(self)`
 - `__eq__(self, other)`
- Details :
<https://stackoverflow.com/questions/4901815/object-of-custom-type-as-dictionary-key>

Dictionary Functions

dict.clear()

Removes all elements of dictionary *dict*.

dict.copy()

Returns a shallow copy of dictionary *dict*.

dict.fromkeys()

Create a new dictionary with keys from seq and values set to *value*.

dict.get(key, default=None)

For key *key*, returns value or default if key not in dictionary.

Dictionary Functions

dict.has_key(key)

Removed, use the **in** operation instead.

dict.items()

Returns a list of *dict*'s (key, value) tuple pairs.

dict.keys()

Returns list of dictionary *dict*'s keys.

dict.setdefault(key, default=None)

Similar to `get()`, but will set `dict[key]=default` if *key* is not already in *dict*.

Dictionary Functions

dict.update(dict2)

Adds dictionary *dict2*'s key-values pairs to *dict*.

dict.values()

Returns list of dictionary *dict*'s values.

MODULES

Modules

- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A **module is a Python object** with arbitrarily named attributes that you can bind and reference.

Example

- The Python code for a module named `aname` normally resides in a file named `aname.py`. Here is an example of a simple module, `support.py`

```
def print_func( par ):  
    print "Hello : ", par  
    return
```

Importing Modules

```
import module1[, module2[,... moduleN]]
```

```
# Import module support  
import support  
  
# Now you can call defined function that module as follows  
support.print_func("Zara")
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening repeatedly, if multiple imports occur.

```
from modname import name1[, name2[, ... nameN]]
```

Executing modules

- Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.
- The code in the module will be executed, just as if you imported it, but with the `__name__` set to "`__main__`"

Executing Modules

```
#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__ == "__main__":
    f=fib(100)
    print(f)
```

Namespaces and Scoping

- Variables are names (identifiers) that map to objects.
- A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

Namespaces and Scoping

- A Python statement can access variables in a *local namespace* and in the *global namespace*.
- If a local and a global variable have the same name, the local variable shadows the global variable.
- In order to assign a value to a global variable within a function, you must first use the `global` statement

GuessTheOutput

```
]def f():  
    A = 5
```

```
A = 1  
print(A)  
f()  
print(A)
```

GuessTheOutput

```
def f():
    global A
    A = 5
```

```
A = 1
print(A)
f()
print(A)
```

FILE I/O

Opening Files

```
file object = open(file_name [, access_mode][, buffering])  
  
    # Open a file  
  
    fo = open("foo.txt", "wb")  
  
    print ("Name of the file: ", fo.name)  
  
    print ("Closed or not : ", fo.closed)  
  
    print ("Opening mode : ", fo.mode)  
  
    fo.close()
```

This produces the following result-

```
Name of the file: foo.txt  
  
Closed or not : False  
  
Opening mode : wb
```

Closing Files

```
fileObject.close();
```

Writing in Files

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opend file
fo.close()
```

Reading Files

```
# Open a file  
fo = open("foo.txt", "r+")  
  
str = fo.read(10)  
  
print ("Read String is : ", str)  
  
# Close opened file  
fo.close()
```

Resources Used

- Tutorials Point
- Google
- Python Documentation

Any Questions??



