Python Identifiers

A Python identifier is a name used to **identify a variable, function, class, module or other object**.

An identifier **starts with a letter A to Z or a to z or an underscore** (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does **not allow punctuation characters** such as **@, $, and %** within identifiers.

Python is a **case sensitive** programming language.

Naming conventions for Python identifiers:

- **Class** names start with an **uppercase** letter. All **other** identifiers start with a **lowercase** letter.
- Starting an identifier with a single leading underscore indicates that the identifier is **private**. Ex.: _a, _func() etc.. _method is just accessible inside the class itself. Even from abc import * will not include this private method
- Starting an identifier with two leading underscores indicates a **strongly private** identifier. Ex.: __a, __func() etc.... it avoids your method to be overridden by a subclass...
- If the identifier also ends with two trailing underscores, the identifier is a **language-defined special name**.

Reserved Words

and	exec	not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

Lines and Indentation

Multi-Line Statements

Quotation in Python **'', "", """ """**
Comments in Python
Using Blank Lines - completely ignored by Python Interpreter

Reading Input from Keyboard

Python 2 has two versions of input functions, **input()** and **raw_input()**. The input() function treats the received data as string if it is included in quotes " or "", otherwise the data is treated as number. The raw_input() used to treat input as string with or without quotes.

In Python 3, raw_input() function is deprecated. Further, the received data is always treated as string.

Integer Division

In Python 2, the result of division of two integers is rounded to the nearest integer. As a result, 3/2 will show 1. In order to obtain a floating-point division, numerator or denominator must be explicitly used as float. Hence, either 3.0/2 or 3/2.0 or 3.0/2.0 will result in 1.5

Python 3 evaluates 3 / 2 as 1.5 by default, which is more intuitive for new programmers.

```
X = 3/2
print (x)
```

Unicode Representation

Python 3 stores strings as Unicode, by default.

Mulitple statments on single line

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

2to3 Utility

```
$2to3 -w area.py
```

The dir( ) Function

import math

content = dir(math)

```
print (content)
```

Standard Data Types

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them.
Python supports three different numerical types −

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

Number Type Conversion

- Type **int(x)** to convert x to a plain 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

abs(x)
cmp(x, y) -> (x>y)-(x<y)
max(x1, x2,...)
min(x1, x2,...)

In Math Module
exp(x)
ceil(x)
floor(x)
log(x)
log10(x)
modf() -returns the fractional and integer parts of x
pow(x, y)
round(x [,n])
sqrt(x)

Python String

Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the.

Update string:

String Formatting Operator

```
print ("My name is {} and id is {} .".format('Nilesh', 70))
print ("My name is %s and id is %d ." % ('Nilesh', 70))
```

Triple Quotes

Raw String

```
print ('C:\\nowhere')    # outputs C:\nowhere
print (r'C:\\nowhere')   # outputs C:\\nowhere
```

Built-in String Methods

```
str.capitalize()
str.center(width[, fillchar])
str.count(sub, start = 0,end = len(string))
str.isa1num()
str.isdigit()
str.islower()
str.join(sequence)
len( str )
```

max (str) # max alphabetical character from the string str

```
str.replace(old, new[, max])
str.split(' ', 2)   # slipt for two instance of delimiter
```

Python List []

Lists are the most versatile of Python's compound data types.

lists are similar to arrays in C.

mutable ordered sequence of elements

One of the differences between them is that all the items belonging to a list can be of different data type. (+) and (*) operates on list

Methods available:

```
marks_regular.append('10')
marks_regular.count(55)
marks_regular.extend( marks_diploma)
marks_regular.index(39)
marks_regular.insert(1, 22)
marks_regular.pop(1)      # removes element and eturn value
marks_regular.remove(1)   # removes element but does not return value
marks_regular.reverse()
marks_regular.sort()
```

Python Tuples ()

Immutable ordered sequence of elements

Difference between lists and tuples are − Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.

Python Dictionary {}

Kind of **hash-table type** and consist of **key-value pairs.**

Any Python type, but are usually numbers or strings.

Simply **unordered**.

Properties of Dictionary Keys

(a) More than one entry per key is not allowed. This means **no duplicate key** is allowed. When duplicate keys are encountered during assignment, the last assignment wins.

(b) Keys must be immutable. This means you **can use strings, numbers or tuples as dictionary keys** but something like ['key'] is not allowed.

Example :

```
dict = {['Name']: 'Zara', 'Age': 7, 45: 23, (22,66) : 'two'}
print (dict)
```

Example : need to import random

```
range(x)
choice(x)
randrange ([start,] stop [,step])
random()                        #0.0 <= r <= 1.0
random.shuffle(list)
random.uniform(5, 10)       #  returns a floating point number between 5 and 10
```

Data Type Conversion

simply use the type-names as a function

S.No.	Function & Description
1	**str(x) :** Converts object x to a string representation.
2	**eval(str) :** Evaluates a string and returns an object.
3	**tuple(s) :** Converts s to a tuple.
4	**list(s) :** Converts s to a list.
5	**set(s) :** Converts s to a set.
6	**dict(d) :** Creates a dictionary. d must be a sequence of (key,value) tuples.
7	**chr(x) :** Converts an integer to a character.
8	**ord(x) :** Converts a single character to its integer value.

```
print(ord('A'))
print(chr(65))

c = 390.8
print(int(c))
```

```
Example :
expr = input("Enter the function(in terms of x):")
x = int(input("Enter the value of x:"))
y = eval(expr)
print("y = {}".format(y))
```

Types of Operator

Python language supports the following types of operators −

- Arithmetic Operators (+ , -, * , / , %, **, //)

Floor Division (//)- The result is floored, i.e., rounded away from zero (towards negative infinity):

- Comparison (Relational) Operators (==, != , > , < , >= , <=)
- Assignment Operators (= , += , -=, *=, /=, %=, **=, //=)
- Logical Operators (AND, OR, NOT)
- Bitwise Operators (&, | , ^ , ~ , << , >>)
- Membership Operators (IN and NOT IN)
- Identity Operators (IS and IS NOT)

Example :

Bitwise Operators

```
a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
print ('a=:',bin(a),'b=:',bin(b))
c = 0

c = a & b;        # 12 = 0000 1100
print ("result of AND is ", c,':',bin(c))
```

Example :

MEMBERSHIP OPERATOR

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ]

print (a in list)
print (b in list)

c=b/a
print (c in list)
```

Identity Operators (IS and IS NOT)

Python built-in **function id()** returns a unique integer as identity of object. Identity operators compare the memory locations of two objects

```
a= 60
b= 10
c= 30 -20
print ( a is b)
```

Decision Making

Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome . Python programming language assumes any **non-zero** and **non-null** values as TRUE, and any **zero** or **null values** as FALSE value.

1] if statements

2] elif statements

3] nested if statements

Syntax : Only if

```
if expression:
   statement(s)
```

Syntax : If and Else

```
if expression:
   statement(s)

else:
   statement(s)
```

Syntax: IfElse

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

Syntax: Nested IfEsle

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

Loops

1] While loop syntax

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

Using else Statement with Loops

Python supports having an **else** statement associated with a loop statement.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Many a times , we need to evaluate last iternation in different way. If we do normally, it could be something like this

WHILE with ELSE

```
Example :
count = 0
while count < 5:
   print (" This is {} iteration".format(count))
   count = count + 1
else:
   print (" This is last iteration")
```

2] for loop syntax

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

Example:

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


for letter in 'DBIT':     # traversal of a string sequence
   print ('Current Letter :', letter)
print()


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

```
Example :
```

```
stud_name = ('ABC', 'LMN', 'XYZ', 'PQR')
for i, stud in enumerate(stud_name,1):
    print (i,stud)
```

FOR with ELSE

Example :

```
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 does not contain even number')
```

NESTED LOOP

Syntax:

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

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

Loop Control Statements (Continue, break, pass)

The **continue** statement in Python returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

The **continue** statement can be used in both *while* and *for* loops.

You continue to the start of next iteration in loop

Iterator

Iterator is an object which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation. In Python, an iterator object implements two methods, **iter()** and **next()**.

Functions

Defining a Function

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. **You can also define parameters inside these parentheses**.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*. **“”” This is docstring“””**
- The code block within every **function starts with a colon (:) and is indented**.

Syntax :

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

Pass agrument by reference or by value ?

Global vs. Local variables

4 Types of argument

1) Required Argument ...Need to follow the order of argument

2) Keyword Argument ... No Need to follow the order of argument

```
def printme(name, str ):
   "This prints a passed string into this function"
   print (str, name)
   return
printme( str = "My string", name = "Nilesh")
```

3) Default Argument .... place default argument at the right hand side...

```
def printme( str, name = 'noname', age ='25' ):
printme( str = "Python", name = "Nilesh")
printme( str = "Python3")
```

4) Variable length Argument ....

```
def printme(*name):
   "This prints a variable passed arguments"
   for var in name:
      print (var)
   return
printme("python")
printme("py", "k","a","i")
```

The Anonymous Functions

use the **lambda** keyword to create small anonymous functions.

- Lambda forms **can take any number of arguments but return just one value in the form of an expression**. They cannot contain commands or multiple expressions.
- An anonymous function **cannot be a direct call to print** because lambda requires an expression.
- Lambda functions have their **own local namespace and cannot access variables other** than those in their parameter list and those in the global namespace.

Syntax:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

```
Example :

sum = lambda arg1, arg2: arg1 + arg2

print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

Modules

Allows you to **logically organize** your Python code.

A **module is a Python object** with arbitrarily named attributes that you can bind and reference.

Simply**, a module is a file consisting of Python code.**

A module **can define functions, classes and variables**. A module can also include runnable code.

The import Statement

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

The from...import Statement

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

Executing Modules as Scripts

The code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__".

It is like constructor ... but not the constructor bcz we have not defined any class over here.. and constructor in class is defined as __init__().

Locating Modules

- The current directory.
- PYTHONPATH
- default path (of linxu/Window)

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.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
- Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.
- The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

Packages in Python

Create it folder --> seit.py , teit.py, beit.py

```
def seit():
print ("Welcome to seit.py")

def teit():
print ("Welcome to teit.py")

def beit():
print ("Welcome to beit.py")
```

```
it/__init__.py
from .seit import seit
from .teit import teit
from .beit import beit



use package in try.py



import it
it.seit()
it.teit()
it.beit()
```

```
Game – secret Code
```

Date & Time

import time # This is required to include time module.

import calendar

ticks = time.time()

print ("Number of ticks since 12:00am, January 1, 1970:", ticks)

print (time.localtime());

```
localtime = time.asctime( time.localtime() )

print ("Local current time :", localtime)


cal = calendar.month(2018, 1)

print (cal)

print (calendar.leapdays(2010,2030))

print (calendar.weekday(2018,1,4))
```

Opening and Closing Files

The open Function

```
file object = open(file_name [, access_mode][, buffering])

access modes:

r,rb,  r+ - read write, rb+, b- binary mode, a -append

If the buffering value is set to 0, no buffering takes place. If the buffering
value is 1, line buffering is performed while accessing a file. If you specify
the buffering value as an integer greater than 1, then buffering action is
performed with the indicated buffer size. If negative, the buffer size is the
system default(default behavior).
```

Example:

```
fo = open("gen.txt", "w")
print ("Name of the file: ", fo.name)
print ("Closed or not : ", fo.closed)
print ("Opening mode : ", fo.mode)
fo.write( "Python is a great language.\nYeah its great!!\n")
fo.close()

fo = open("gen.txt", "r+")
str = fo.read()
print ("Read String is : ", str)
# Check current position
position = fo.tell()
print ("Current file position : ", position)
fo.write("This is new line")
fo.seek(6,0)
fo.write(" This is second new line ")
fo.close()



fo = open("gen.txt", "a")
fo.write("This is new line")
fo.close()
```

Renaming and Deleting Files

```
import os
os.rename( "test1.txt", "test2.txt" )
os.remove("text2.txt")
os.mkdir("newdir")
```

```
os.getcwd()    # displays the current working directory

os.chdir("/home/dbit/")   #method to change the current directory

os.rmdir('dirname')
```

Exceptions Handling

There are approx 29 standard exception defined in python. Few of them are

Exception,IndentationError, SyntaxError, StopIteration, SystemExit, ArithmeticError

OverflowError, ZeroDivisonError, EOFError, NameError, IOError

USER DEFINED EXCEPTION

```
try:

   You do your operations here
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

```
import time
try:
    fo = open("gen.txt", "w")
    fo.write( "Python is a great language.\nYeah its great!!\n")
    print("READ:",fo.read())
except IOError:
    print ("Error: can\'t find file or read data")
    localtime=time.time()
    fo.write ("Someone tried to read data at {} ".format(localtime))
else:
    print ("Written content in the file successfully")
fo.close()
```

```
# can use finally  instead of else..but not both
```

Else shld be written with except clause... It gives error when we use else without except.

```
import time
try:
    fo = open("gen.txt", "w")
    fo.write( "Python is a great language.\nYeah its great!!\n")
    print("READ:",fo.read())
else:
    print ("Written content in the file successfully")
fo.close()
```

we should nt write finally with except clause. It must execute, whether the try-block raised an exception or not

```
import time
try:
    fo = open("gen.txt", "w")
    fo.write( "Python is a great language.\nYeah its great!!\n")
    print("READ:",fo.read())
finally:
    print ("Written content in the file successfully")
fo.close()
```

Object Oriented

```
class Emp:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Emp.empCount += 1

   def __str__(self):
       print(self.name, self.salary)       // str returns and not print

   def displayCount(self):
     print ("Total Employee %d" , Emp.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)
```

```
e1 = Emp("abc", 7550)
e2 = Emp("lmn", 3590)

e1.displayEmployee()
e2.displayEmployee()
print ("Total Employee %d" % Emp.empCount)
print (e1)
```

Built-In Class Attributes

```
print ("Employee.__doc__:", Emp.__doc__)
print ("Employee.__name__:", Emp.__name__)
print ("Employee.__module__:", Emp.__module__)
print ("Employee.__bases__:", Emp.__bases__)
print ("Employee.__dict__:", Emp.__dict__ )
```

Accessing object

```
print(hasattr(e1, 'salary'))
print(getattr(e1, 'salary'))
print(setattr(e1, 'salary', 7000))
print(getattr(e1, 'salary'))
delattr(e1, 'salary')
print(hasattr(e1, 'salary'))
```

Destroying Objects (Garbage Collection)

Maintains ref count.... When an object's reference count reaches zero, Python collects it automatically.

Can be accessed using id(obj)

Class Inheritance

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print ("Calling parent constructor")

   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)

class Child(Parent):          # define child class, u can add multiple parent class here
   def __init__(self):
```

```
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

Base Overloading Methods

`__init__ ( self [,args...] )`
`__del__( self )` Destructor, deletes an object
`__str__( self )` Printable string representation

Data Hiding

You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

```
class Emp:
  __empCount = 0

  def count(self):
    self.__empCount += 1
    print (self.__empCount)

c = Emp()
c.count()
c.count()
print (c.__empCount)
```

DATABASE

You need Python-MySQL Package to connect Python program with MySQL.
1] Download "**PyMySQL-master.zip**" from https://github.com/PyMySQL/PyMySQL/ and save in directory called let say "connector". Extract the zip file.
2] In you Atom Editor and open terminal... change the current present working directory to "connector" directory.
3] Now type "python setyp.py install" on terminal. It will get install.

CREATE DB

```
# Drop table if it already exist using execute() method.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement
sql = """CREATE TABLE EMPLOYEE (
   FIRST_NAME  CHAR(20) NOT NULL,
   LAST_NAME  CHAR(20),
   AGE INT,
   SEX CHAR(1),
   INCOME FLOAT )"""

cursor.execute(sql)
```

INSERT

```
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
   LAST_NAME, AGE, SEX, INCOME)
   VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
```

OR

```
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
   LAST_NAME, AGE, SEX, INCOME) \
   VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
   ('Mac', 'Mohan', 20, 'M', 2000)
```

READ

```
import pymysql

db = pymysql.connect("localhost","root","root","ost")
cursor = db.cursor()

sql = "SELECT * FROM ost_submission"
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Fetch all the rows in a list of lists.
   results = cursor.fetchall()
   for row in results:
      seat = row[2]
      name = row[3]
      filepath = row[4]
      # Now print fetched result
      print ("seat = %s,name = %s,filepath = %s" % \
             (seat, name, filepath))
except:
   print ("Error: unable to fetch data")

db.close()
```