Python3 Programming Language

Presentation Outline

- Python Overview
- Python Data Types
- Python Control Structures
- Python Input\output
- Python Functions
- Python File Handling
- Python Exception [illegible]
- Python Modules
- Python Classes
- Python vs. Java Examples
- Python Useful Tools
- Who uses Python?

Python Overview

What is Python?

Python is a high-level programming language which is:

- **Interpreted:** Python is processed at runtime by the interpreter.
- **Interactive:** You can use a Python prompt and interact with the interpreter directly to write your programs.
- **Object-Oriented:** Python supports Object-Oriented technique of programming.
- **Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications.

History of Python

- Python was conceptualized by **Guido Van Rossum** in the late **1980s**.
- Rossum published the first version of Python code (0.9.0) in February **1991** at the CWI (Centrum Wiskunde & Informatica) in the Netherlands , Amsterdam.
- Python is derived from **ABC** programming language, which is a general-purpose programming language that had been developed at the CWI.
- Rossum chose the name "**Python**", since he was a big fan of Monty Python's Flying Circus.
- Python is now maintained by a core development team at the institute, although Rossum still holds a vital role in directing its progress.

https://en.wikipedia.org/wiki/Guido_van_Rossum#/media/File:Guido_van_Rossum_OSCON_2006.jpg

Python Versions

Release dates for the major and minor versions:

- **Python 1.0** - January **1994**
  - Python 1.5 - December 31, 1997
  - Python 1.6 - September 5, 2000
- **Python 2.0** - October 16, **2000**
  - Python 2.1 - April 17, 2001
  - Python 2.2 - December 21, 2001
  - Python 2.3 - July 29, 2003
  - Python 2.4 - November 30, 2004
  - Python 2.5 - September 19, 2006
  - Python 2.6 - October 1, 2008
  - Python 2.7 - July 3, 2010

Python Versions

Release dates for the major and minor versions:

- **Python 3.0** - December 3, **2008**
  - Python 3.1 - June 27, 2009
  - Python 3.2 - February 20, 2011
  - Python 3.3 - September 29, 2012
  - Python 3.4 - March 16, 2014
  - Python 3.5 - September 13, 2015

Key Changes in Python 3.0

- Python 2's print statement has been replaced by the **print()** function.

Old:
```
print 'Hello, World!'
```

New:
```
print('Hello, World!')
```

- There is only one integer type left, **int.**
- Some methods such as map() and filter( ) return **iterator** objects in Python 3 instead of lists in Python 2.
- In Python 3, a TypeError is raised as warning if we try to compare unorderable types. e.g. 1 < ' ', 0 > None are ***no*** longer valid
- Python 3 provides Unicode (**utf-8**) strings while Python 2 has ASCII str( ) types and separate unicode( ).
- A new built-in string formatting method **format()** replaces the **%** string formatting operator.

Key Changes in Python 3.0

- In Python 3, we should enclose the exception argument in parentheses.

Old:
```
raise IOError, "file error"
```

New:
```
raise IOError("file error")
```

- In Python 3, we have to use the **as** keyword now in the handling of exceptions.

Old:
```
try:
    ...
except NameError, err:
    ...
```

New:
```
try:
    ...
except NameError as err:
    ...
```

- The division of two integers returns a **float** instead of an integer. "**//**" can be used to have the "old" behavior.

Python Features

- **Easy to learn, easy to read and easy to maintain.**
- **Portable**: It can run on various hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter.
- **Scalable:** Python provides a good structure and support for large programs.
- Python has support for an **interactive mode** of testing and debugging.
- Python has a broad standard **library** cross-platform.
- Everything in Python is an **object**: variables, functions, even code. Every object has an ID, a type, and a value.

```
>>> x=36
>>> id(x)
4297539008
>>> type(x)
<class 'int'>
```

More Features ..

- Python provides interfaces to all major commercial **databases**.
- Python supports functional and structured programming methods as well as **OOP**.
- Python provides very high-level **dynamic** data types and supports dynamic type checking.
- Python supports **GUI** applications
- Python supports automatic **garbage collection**.
- Python can be easily **integrated** with C, C++, and Java.

Python Syntax

Basic Syntax

Indentation is used in Python to delimit blocks. The number of spaces is variable, but all statements within the same block must be indented the same amount.

- The header line for compound statements, such as if, while, def, and class should be terminated with a colon (:)
- The semicolon (;) is optional at the end of statement.

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

- Printing to the Screen:

```
print ("Hello, Python!")
```

- Reading Keyboard Input:

```
name = input("Enter your name: ")
```

- **Comments**
  - Single line:

```
# This is a comment.
```

  - Multiple lines:

```
'''
print("We are in a comment")
print ("We are still in a comment")
'''
```

- Python files have extension .py

Variables

- Python is dynamically typed. You do not need to declare variables!
- The declaration happens automatically when you assign a value to a variable.
- Variables can change type, simply by assigning them a new value of a different type.
- Python allows you to assign a single value to several variables simultaneously.
- You can also assign multiple objects to multiple variables.

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
z = None               # A null value
```

```
x = 1
x = "string value"
```

```
a = b = c = 1
```

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

Python Data Types

Numbers

- Numbers are **Immutable** objects in Python that cannot change their values.
- There are three built-in data types for numbers in Python3:
  - Integer (int)
  - Floating-point numbers (float)
  - Complex numbers: *<real part>* + *<imaginary part>*j (not used much in Python programming)
- **Common Number Functions**

Function	Description
int(x)	to convert x to an integer
float(x)	to convert x to a floating-point number
abs(x)	The absolute value of x
cmp(x,y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: e^x
log(x)	The natural logarithm of x, for x> 0
pow(x,y)	The value of x**y
sqrt(x)	The square root of x for x > 0

Strings

- Python Strings are **Immutable** objects that cannot change their values.

```
>>> str= "strings are immutable!"
>>> str[0]="S"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- You can update an existing string by (re)assigning a variable to another string.
- Python *does not* support a character type; these are treated as strings of length one.
- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals.

```
name1 = "sample string"
name2 = 'another sample string'
name3 = """a multiline
  string example"""
```

- String indexes starting at **0** in the beginning of the string and working their way from **-1** at the end.

Strings

- **String Formatting**

```
>>> num = 6
>>> str= "I have {} books!".format(num)
>>> print(str)
I have 6 books!
```

- **Common String Operators**

Assume string variable **a** holds 'Hello' and variable **b** holds

Operator	Description	Example
+	**Concatenation** - Adds values on either side of the operator	a + b will give HelloPython
*****	**Repetition** - Creates new strings, concatenating multiple copies of the same string	a*2 will give HelloHello
[ ]	**Slice** - Gives the character from the given index	a[1] will give e a[-1] will give o
[:]	**Range Slice** - Gives the characters from the given range	a[1:4] will give ell
in	**Membership** - Returns true if a character exists in the given string	‘H’ in a will give True

Strings

- **Common String Methods**

Method	Description
str.**count**(sub, beg= 0,end=len(str))	Counts how many times sub occurs in string or in a substring of string if starting index beg and ending index end are given.
str.**isalpha**()	Returns True if string has at least 1 character and all characters are alphanumeric and False otherwise.
str.**isdigit**()	Returns True if string contains only digits and False otherwise.
str.**lower**()	Converts all uppercase letters in string to lowercase.
str.**upper**()	Converts lowercase letters in string to uppercase.
str.**replace**(old, new)	Replaces all occurrences of old in string with new.
str.**split**(str=' ')	Splits string according to delimiter str (space if not provided) and returns list of substrings.
str.**strip**()	Removes all leading and trailing whitespace of string.
str.**title**()	Returns "titlecased" version of string.

- **Common String Functions**

str(x) :to convert x to a string

len(string):gives the total length of the string

Lists

- A list in Python is an **ordered** group of items or elements, and these list elements *don't have* to be of the same type.
- Python Lists are **mutable** objects that can change their values.
- A list contains items separated by *commas* and enclosed within *square brackets*.
- List indexes like strings starting at **0** in the beginning of the list and working their way from **-1** at the end.
- Similar to strings, Lists operations include **slicing** ([ ] and [:]) , **concatenation** (+), **repetition** (*), and **membership** (in).
- This example shows how to *access*, *update* and *delete* list elements:

```
>>> list = ['physics', 'chemistry', 1997, 2000,2015]
>>> print (list[0])        access
physics
>>> print (list[1:4])      slice
['chemistry', 1997, 2000]
>>> list[2] = 1999         update
>>> print (list[2])
1999
>>> del (list[4])          delete
>>> print (list)
['physics', 'chemistry', 1999, 2000]
```

Lists

- Lists can have sublists as elements and these sublists may contain other sublists as well.

```
>>> person = [["Tahani", "Nasser"],["Boulder","CO"]]
>>> first_name = person[0][0]
>>> city = person[1][0]
>>> print(first_name+" lives in "+ city)
Tahani lives in Boulder
```

- **Common List Functions**

Function	Description
cmp(list1, list2)	Compares elements of both lists.
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(tuple)	Converts a tuple into list.

Lists

- **Common List Methods**

Method	Description
list.**append**(obj)	Appends object obj to list
list.**insert**(index, obj)	Inserts object obj into list at offset index
list.**count**(obj)	Returns count of how many times obj occurs in list
list.**index**(obj)	Returns the lowest index in list that obj appears
list.**remove**(obj)	Removes object obj from list
list.**reverse**()	Reverses objects of list in place
list.**sort**()	Sorts objects of list in place

- **List Comprehensions**

  Each list comprehension consists of an **expression** followed by a **for** clause.

```
>>> a = [1, 2, 3]
>>> [x ** 2 for x in a]  List comprehension
[1, 4, 9]
>>> z = [x + 1 for x in [x ** 2 for x in a]]
>>> z
[2, 5, 10]
```

Tuples

- Python Tuples are **Immutable** objects that cannot be changed once they have been created.
- A tuple contains items separated by *commas* and enclosed in *parentheses* instead of square brackets.

```
>>> t = ("tuples", "are", "immutable")
>>> t[0]  ->access
'tuples'
>>> t[0]="assignments to elements are not possible"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

->No update

- You can update an existing tuple by (re)assigning a variable to another tuple.
- Tuples are **faster** than lists and **protect** your data against accidental changes to these data.
- The rules for tuple **indices** are the same as for lists and they have the same **operations,**
  functions as well.
- To write a tuple containing a single value, you have to include a *comma*, even though there is only one value. e.g. t = (3,)

Dictionary

- Python's dictionaries are kind of hash table type which consist of **key-value** pairs
  of **unordered** elements.
  - **Keys** : must be immutable data types ,usually numbers or strings.
  - **Values** : can be any arbitrary Python object.
- Python Dictionaries are **mutable** objects that can change their values.
- A dictionary is enclosed by *curly braces* ({ }), the items are separated by *commas*, and each key is separated from its value by a *colon* (:).
- Dictionary's values can be assigned and accessed using square braces ([]) with a key to obtain its value.

Dictionary

- This example shows how to *access*, *update* and *delete* dictionary elements:

```
dict = {'Name': 'Jood', 'Age': 9, 'Grade': '5th'}
# Access Dictionary
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
print(dict.keys())                  #list of dict's keys
print(dict.values())                #list of dict's values
print(dict.items())                 #list of dict's tuple pairs

# Update Dictionary
dict['Age'] = 10                    # update existing entry
dict['School'] = "Fireside Elementary School"  # Add new entry

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])

# Delete Dictionary
del dict['Name']       # remove entry with key 'Name'
print (dict)
dict.clear()           # remove all entries in dict
print (dict)
del dict               # delete entire dictionary
print (dict)
```

- The output:

```
dict['Name']:  Jood
dict['Age']:  9
dict_keys(['Name', 'Grade', 'Age'])
dict_values(['Jood', '5th', 9])
dict_items([('Name', 'Jood'), ('Grade', '5th'), ('Age', 9)])
dict['Age']:  10
dict['School']:  Fireside Elementary School
{'School': 'Fireside Elementary School', 'Grade': '5th', 'Age': 10}
{}
<class 'dict'>
```

Dictionary

- **Common Dictionary Functions**
  - **cmp**(dict1, dict2) : compares elements of both dict.
  - **len**(dict) : gives the total number of (key, value) pairs in the dictionary.

- **Common Dictionary Methods**

Method	Description
dict.**keys**()	Returns list of dict's keys
dict.**values**()	Returns list of *dict*'s values
dict.**items**()	Returns a list of *dict*'s (key, value) tuple pairs
dict.**get**(key, default=None)	For key, returns value or default if key not in dict
dict.**has_key**(key)	Returns *True* if key in *dict*, *False* otherwise
dict.**update**(dict2)	Adds *dict2*'s key-values pairs to *dict*
dict.**clear**()	Removes all elements of *dict*

Python Control Structures

Conditionals

- In Python, **True** and **False** are Boolean objects of class '**bool**' and they are **immutable**.
- Python assumes any **non-zero** and **non-null** values as **True**, otherwise it is **False** value.
- Python *does not* provide switch or case statements as in other languages.
- **Syntax:**

if Statement

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

if..else Statement

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

if..elif..else Statement

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

- **Example:**

```
x = int(input("Please enter an integer: "))
if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

Conditionals

Using the conditional expression

Another type of conditional structure in Python, which is very convenient and easy to read.

```
a, b = 4, 5

if a < b:
    x = 'smaller'
else:
    x = 'bigger'

print (x)
```

```
x = 'smaller' if a < b else 'bigger'
```

Loops

The For Loop

```
# First Example
for letter in 'Python':
   print ('Current Letter :', letter)

# Second Example
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:
   print ('Current fruit :', fruit)

# Third Example (Iterating by Sequence Index)
food = ['pizza', 'steak',  'rice']
for index in range(len( food )):     # range(3) iterates between 0 to 2
   print ('Current food :', food[index])
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Current food : pizza
Current food : steak
Current food : rice
```

The while Loop

```
count = 0
while (count < 5):
   print ('The count is:', count)
   count = count + 1
```

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
```

Loops

Loop Control Statements

Break: Terminates the loop statement and transfers execution to the statement immediately following the loop.

```
for letter in 'Python':
   if letter == 'h':
      break
   print ('Current Letter :', letter)
```

```
Current Letter : P
Current Letter : y
Current Letter : t
```

- **continue**:Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

```
for letter in 'Python':
   if letter == 'h':
      continue
   print ('Current Letter :', letter)
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

- **pass**:Used when a statement is required syntactically but you do not want any command or code to execute.

```
for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :', letter)
```

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
```

Python Functions

Functions

- **Function Syntax**

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

- **Function Arguments**

You can call a function by using any of the following types of arguments:

- **Required arguments**: the arguments passed to the function in correct positional order.
- **Keyword arguments**: the function call identifies the arguments by the parameter names.
- **Default arguments**: the argument has a default value in the function declaration used when the value is not provided in the function call.

```
def func( name, age ):
....
func("Alex", 50)
```

```
def func( name, age ):
....
func( age=50, name="Alex" )
```

```
def func( name, age = 35 ):
...
func( "Alex" )
```

Functions

- **Variable-length arguments:** This used when you need to process unspecified additional arguments. An asterisk (*) is placed before the variable name in the function declaration.

```
def printinfo( arg1, *vartuple ):
   print ("Output is: ")
   print (arg1)
   for var in vartuple:
      print (var)
   return

printinfo( 5 )
printinfo( 10, 20, 30 )
```

```
Output is:
5
Output is:
10
20
30
```

Python File Handling

File Handling

- **File opening** fileObject = **open**(file_name [, access_mode][, buffering])

  Common access modes:

  - "**r**" opens a file for reading only.
  - "**w**" opens a file for writing only. Overwrites the file if the file exists. Otherwise, it creates a new file.
  - "**a**" opens a file for appending. If the file does not exist, it creates a new file for writing.

- **Closing a file** fileObject.**close**(
)

  The close() method flushes any unwritten information and closes the file object.

File Handling

- **Reading a file** fileObject.**read**([count])
  - The ***read()*** method reads the whole file at once.
  - The **readline**() method reads one line each time from the file.
  - The ***readlines()*** method reads all lines from the file in a list.

- **Writing in a file** fileObject.**write**(string)

  The write() method writes any string to an open file.

Python Exception Handling

Exception Handling

- **Common Exceptions in Python:**

  NameError - TypeError - IndexError - KeyError - Exception

- **Exception Handling Syntax:**

```
try:
    statements to be inside try clause
    statement2
    statement3
    ...
except ExceptionName:
    statements to evaluated in case of ExceptionName happens
```

- An empty except statement can catch any exception.
- ***finally*** clause: always executed before finishing try statements.

```
try:
    fobj = open("hello.txt", "w")
    res = 12 / 0
except ZeroDivisionError:
    print("We have an error in division")
finally:
    fobj.close()
    print("Closing the file object.")
```

⇨

```
We have an error in division
Closing the file object.
```

Python Modules

Modules

- A module is a file consisting of Python code that can define functions, classes and variables.
- A module allows you to organize your code by grouping related code which makes the code easier to understand and use.
- You can use any Python source file as a module by executing an ***import*** statement

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

- Python's ***from*** statement lets you import **specific** attributes from a module into the current namespace.

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

- ***import*** * statement can be used to import **all** names from a module into the current namespace

```
from modname import *
```

Python Object Oriented

Python Classes

```
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)

"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)
```

 Class variable

 Class constructor

Output

```
Name :  Zara , Salary:  2000
Name :  Manni , Salary:  5000
Total Employee 2
```

Python Classes

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

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

- **Data Hiding** You need to name attributes with *a double underscore prefix*, and those attributes then are not be directly visible to outsiders.

```
self.__name = name
self.__salary = salary
```

Class Inheritance

```
class Person:

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

    def get_details(self):
        "Returns a string containing name of the person"
        return self.name


class Student(Person):

    def __init__(self, name, branch, year):
        Person.__init__(self, name)
        self.branch = branch
        self.year = year

    def get_details(self):
        "Returns a string containing student's details."
        return "%s studies %s and is in %s year." % (self.name, self.branch, self.year)


person1 = Person('Alex')
student1 = Student('Jake', 'CSE', 2015)

print(person1.get_details())
print(student1.get_details())
```

```
Alex
Jake studies CSE and is in 2015 year.
```

Python vs. Java

Code Examples

Python vs. Java

- Hello World

Java

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

Python

```python
print ("hello world")
```

- String Operations

Java

```java
public static void main(String[] args) {
    String test = "compare Java with Python";
    for(String a : test.split(" "))
    System.out.print(a);
}
```

Python

```python
a="compare Python with Java"
print (a.split())
```

Python vs. Java

- Collections

Java

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("a");
        al.add("b");
        al.add("c");
        System.out.println(al);
    }
}
```

Python

```python
aList = []
aList.append("a")
aList.append("b")
aList.append("c")
print (aList)
```

Python vs. Java

- Class and Inheritance

Java

```java
class Animal{
    private String name;
    public Animal(String name){
        this.name = name;
    }
    public void saySomething(){
        System.out.println("I am " + name);
    }
}

class Dog extends Animal{
    public Dog(String name) {
        super(name);
    }
    public void saySomething(){
        System.out.println("I can bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Chiwawa");
        dog.saySomething();

    }
}
```

Python

```python
class Animal():

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

        def saySomething(self):
            print ("I am " + self.name)

class Dog(Animal):
        def saySomething(self):
            print ("I am "+ self.name \
            + ", and I can bark")

dog = Dog("Chiwawa")
dog.saySomething()
```

```
I am Chiwawa, and I can bark
```

Python Useful Tools

Useful Tools

- **Python IDEs**
  - Vim
  - Eclipse with PyDev
  - Sublime Text
  - Emacs
  - Komodo Edit
  - PyCharm

Useful Tools

- **Python Web Frameworks**
  - Django
  - Flask
  - Pylons
  - Pyramid
  - TurboGears
  - Web2py

Who Uses Python?

Organizations Use Python

- **Web Development** :Google, Yahoo
- **Games** :Battlefield 2, Crystal Space
- **Graphics** :Walt Disney Feature Animation, Blender 3D
- **Science** :National Weather Service, NASA, Applied Maths
- **Software Development** :Nokia, Red Hat, IBM