

Introduction to Python Programming



Presented by
Dr. Helen Josephine V L
Associate professor
Dept of MCA, CMRIT

Day 1 - Agenda



Python Overview



Python Basics



Collections



Functions, Modules

What is Python

Python is

- General purpose
- High level interpreted language
- with easy syntax and
- dynamic semantics



Created by Guido Van Rassum in 1989 at
National Institute for Mathematics and
Computer Science, Netherland

Why Python so Popular

Usage in Big Companies- Google

Data Analysis & Scientific Research

Library and Support

Easy

Open Source

Application



Python Features



Features

- Simple and easy to learn
- Freeware and Open source
- Interpreted
- Dynamically Typed
- Extensible
- Embedded
- Extensive library



Career Opportunities

01

Web Development & Frameworks

02

Game Development

03

Big Data Analytics

04

Web Testing

05

AI / Data Science

06

IOT – Smart Devices



Where do I start with Python?

Variable, Data Type, Operators

Flow Control

Collections

Funtcions, Methods

Object Oriented Programming

Practice Programming

How to run

Python : <https://www.python.org/downloads/>

Anaconda : <https://www.anaconda.com/products/individual>



- **Anaconda** is a free distribution of the **Python** programming language for
- large-scale data processing,
- predictive analytics, and
- scientific computing,
- that aims to simplify package management and deployment

How to run

Jupyter Notebook



- **Jupyter Notebook** is an open-source web application
 - allows to create and share documents
 - contains
 - live code
 - equations
 - visualizations
 - narrative text

How to run

Google Colab



- Free cloud service
- Supports free GPU
- Improve Python coding skills
- Develop ML & DL applications
 - Keras
 - TensorFlow
 - PyTorch
 - OpenCV.



Python Basics

Print Statement

```
print("Welcome to this Session")
```

Output : Welcome to this Session

```
a=10
```

```
print(a)
```

Output : 10

```
x="Python"
```

```
print(x)
```

Output : Python



Python Basics

User Input Statement

```
Name = input()
print(Name, "! Welcome to this Session")
```

Akil

Output :

Akil! Welcome to this Session

```
Name = input("Enter the Name : ")
print(Name, "! Welcome to this Session")
```

Enter the Name : Akil

Output :

Akil! Welcome to this Session



Python Basics

User Input Statement

```
internal_mark = int(input("Enter Internal Mark : "))  
external_mark = int(input("Enter External Mark : "))  
tot_mark = internal_mark + external_mark  
print("Total Mark : ",tot_mark)
```

Output :

Enter Internal Mark : 38

Enter External Mark : 55

Total Mark : 93

Variables

- An identifier for the data and it holds data in your program
- Is a location (or set of locations) in memory where a value can be stored
- A quantity that can change during program execution
- **No declaration of variables**
- **Data type of a variable can change during program execution compared to other strongly typed languages such as Java, C++, C**

```
customer_id = 101
customer_name = "John"
bill_amount = 675.45
x = 5.3 + 0.9j
print(customer_id, customer_name, bill_amount)
print(x.real)
print(x.imag + 3)
```

Integer
String
Floating-point
complex number
#prints 101 John 675.45
#prints 5.3
#prints 3.9

Operators (1 of 6)

- Used to perform specific operations on one or more operands (or variables) and provide a result





Conditional Execution

Control Flow

Syntax:

```
if condition1:  
    statement(s)  
else:  
    statement(s)
```

```
x = 34 - 23           # A comment.  
y = "Good"           # Another one.  
z = 3.45  
if z == 3.45 or y == "Session":  
    x = x + 1  
    y = y + "Session" # String concat.  
print(x)  
print(y)
```

Syntax:

```
if condition1:  
    statement(s)  
elif condition2:  
    statement(s)  
else:  
    statement(s)
```

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

Looping Structure

Iterative Statements

Loop Statements : Allows us to execute a statement or group of statements multiple times

- While Loop
- For Loop
- Range

Loop Control Statements : Are used to change flow of execution from its normal sequence

- Break
- Continue
- Pass

Looping Structure

while Loop

Iterative Statements

– **while loop:**

- Repeats a statement or group of statements while a given condition is TRUE.
- Tests the condition before executing the loop body.

Example:

```
n = 5
result = 0
counter = 1
while counter <= n:
    result = result + counter
    counter += 1
print("Sum of 1 until %d: %d" % (n, result))
```

Syntax:

```
while condition:
    statement(s)
```

Output:

```
Sum of 1 until 5: 15
```

Looping Structure

for Loop

Iterative Statements

– for loop:

- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax:

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

Example:

```
for counter in 1,2,'Sita', 7,'Ram',5:  
    print(counter)
```

Output:

```
1  
2  
Sita  
7  
Ram  
5
```

Looping Structure

range function in loop

Iterative Statements

– range function in loops

- Used in case the need is to iterate over a specific number of times within a given range in steps/intervals mentioned

Syntax: `range(lower limit, upper limit, Increment/decrement by)`

Loop	Output	Remarks
<code>for value in range(1,6): print(value)</code>	1 2 3 4 5	Prints all the values in given range exclusive of upper limit
<code>for value in range(0,6,2): print(value)</code>	0 2 4	Prints values in given range in increments of 2
<code>for value in range(6,1,-2): print(value)</code>	6 4 2	Prints values in given range in decrements of 2
<code>for ch in "Hello World": print(ch.upper())</code>	H E L L O W O R L D	Prints all the characters in the string converting them to upper case



Looping Structure

break

The break statement ends the current loop and jumps to the statement immediately following the loop

It is like a loop test that can happen anywhere in the body of the loop

```
while True:  
    line = input('> ')  
    if line == 'done' :  
        break  
    print(line)  
print('Done!')
```

```
> hello there  
hello there  
> finished  
finished  
> done  
Done!
```



Looping Structure

continue

The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

Q & A

Question
and
Answer



Collections in Python

- Dynamic way of organizing data in memory
- Some of the Collections available in Python are:
 - Strings
 - List
 - Tuples
 - Sets
 - Dictionaries

String



String Operators and Functions

```
# defining strings in Python
# all of the following are equivalent
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)

my_string = """Hello"""
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
    the world of Python"""
print(my_string)
```

```
Hello
Hello
Hello
Hello, welcome to
    the world of Python
```

String

String Operators and Functions

```
#Accessing string characters in Python
```

```
str = 'programiz'  
print('str = ', str)
```

```
#first character
```

```
print('str[0] = ', str[0])
```

```
#last character
```

```
print('str[-1] = ', str[-1])
```

```
#slicing 2nd to 5th character
```

```
print('str[1:5] = ', str[1:5])
```

```
#slicing 6th to 2nd last character
```

```
print('str[5:-2] = ', str[5:-2])
```

```
str = programiz
```

```
str[0] = p
```

```
str[-1] = z
```

```
str[1:5] = rogr
```

```
str[5:-2] = am
```

String

String Operators and Functions

- **Concatenation**
 - Strings can be concatenated with '+' operator
 - "Hello" + "World" will result in **HelloWorld**
- **Repetition**
 - Repeated concatenation of string can be done using asterisk operator "*" **"**"**
 - "Hello" * 3 will result in ***HelloHelloHello***
- **Indexing**
 - "Python"[0] will result in **"P"**
- **Slicing**
 - Substrings are created using two indices in a square bracket separated by a ':'
 - "Python"[2:4] will result in **"th"**
- **Size**
 - prints length of string
 - len("Python") will result in **6**

List

List Creation

- An ordered group of sequences enclosed inside square brackets [], separated by ,
- It can have any number of items and may be of different types (integer, float, string etc.).
- Lists are mutable – can change existing values

```
my_list = []
```

```
# list of integers
```

```
my_list = [1, 2, 3]
```

```
# list with mixed data types
```

```
my_list = [1, "Hello", 3.4]
```

List

List Notations and Examples

List Example	Description
<code>[]</code>	An empty list
<code>[1, 3, 7, 8, 9, 9]</code>	A list of integers
<code>[7575, "Shyam", 25067.56]</code>	A list of mixed data types
<code>["Bangalore", "Bhubaneswar", "Chandigarh", "Chennai", "Hyderabad", "Mangalore", "Mysore", "Pune", "Trivandrum"]</code>	A list of Strings
<code>[[7575, "John", 25067.56], [7531, "Joe", 56023.2], [7821, "Jill", 43565.23]]</code>	A nested list
<code>["India", ["Karnataka", ["Mysore", [GEC1, GEC2]]]]</code>	A deeply nested list



List

List Notations and Examples

Python Expression	Result	Operation
<code>len([4, 5, 6])</code>	3	Length
<code>[1, 3, 7] + [8, 9, 9]</code>	<code>[1, 3, 7, 8, 9, 9]</code>	Concatenation
<code>['Hello'] * 4</code>	<code>['Hello', 'Hello', 'Hello', 'Hello']</code>	Repetition
<code>7 in [1, 3, 7]</code>	True	Membership
<code>for n in [1, 3, 7]: print(n)</code>	1 3 7	Iteration
<code>n = [1, 3, 7]</code> <code>print(n[2])</code>	7	Indexing: Offset starts at 0
<code>n = [1, 3, 7]</code> <code>print(n[-2])</code>	3	Negative slicing: Count from right
<code>n = [1, 3, 7]</code> <code>print(n[1:])</code>	<code>[3, 7]</code>	Slicing



List

List Creation

- ❑ access a range of items in a list by using the slicing operator :(colon)

```
my_list = ['p','y','t','h','o','n','p','r','o']
```

```
# elements 3rd to 5th
```

```
print(my_list[2:5])
```

```
# elements beginning to 4th
```

```
print(my_list[:-5])
```

```
# elements 6th to end
```

```
print(my_list[5:])
```

```
# elements beginning to end
```

```
print(my_list[:])
```

```
['t', 'h', 'o']
```

```
['p', 'y', 't', 'h']
```

```
['n', 'p', 'r', 'o']
```

```
['p', 'y', 't', 'h', 'o', 'n', 'p', 'r', 'o']
```



Tuples

- ❑ lists, strings, tuples: examples of sequence type, separated by ,
- ❑ Immutable – can not change the value

```
>>> t = 123, 543, 'bar'  
>>> t[0]  
123  
>>> t  
(123, 543, 'bar')
```

Tuples may be nested

```
>>> u = t, (1,2)  
>>> u  
((123, 542, 'bar'), (1,2))
```

```
my_tuple = 3, 4.6, "dog"  
print(my_tuple)
```

```
# tuple unpacking is also  
possible
```

```
a, b, c = my_tuple
```

```
print(a)    # 3  
print(b)    # 4.6  
print(c)    # dog
```

Tuples

Access Tuple Elements

Indexing

Negative Indexing

Slicing

Changing a Tuple

Deleting a Tuple

Tuple Methods - count, index

Tuple Membership Test - in

Iterating

Set

- An un-ordered collection of unique elements
- Are lists with no index value and no duplicate entries
- Can be used to identify unique words used in a paragraph

Syntax:

```
set1 = {}           #Creation of empty set  
set2 = {"John"}    #Set with an element
```

Example:

```
s1 = set("my name is John and John is my name".split())  
s1 = {'is', 'and', 'my', 'name', 'John'}
```

- Operations like intersection, difference, union, etc. can be performed on sets

Dictionary

- A list of elements with key and value pairs(separated by symbol :) inside curly braces.
- Keys are used instead of indexes
- Keys are used to access elements in dictionary and keys can be of type – strings, number list, etc
- Dictionaries are mutable, i.e it is possible to add, modify and delete key-value pairs

Syntax:

```
phonebook = {} #Creation of empty Dictionary  
phonebook={"John":938477565} #Dictionary with one key-value pair  
phonebook={"John":938477565, "Jill":938547565} #2 key-value pairs
```



Dictionary

```
#empty dictionary
```

```
my_dict = {}
```

```
# dictionary with integer keys
```

```
my_dict = {1: 'apple', 2: 'ball'}
```

```
# dictionary with mixed keys
```

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

```
# using dict()
```

```
my_dict = dict({1:'apple', 2:'ball'})
```

```
# from sequence having each item as a pair
```

```
my_dict = dict([(1,'apple'), (2,'ball')])
```

Dictionary

```
# get vs [] for retrieving elements  
my_dict = {'name': 'Jack', 'age': 26}  
  
# Output: Jack  
print(my_dict['name'])  
  
# Output: 26  
print(my_dict.get('age'))
```

Output

```
Jack  
26
```

Dictionary

Changing and adding Dictionary Elements

```
my_dict = {'name': 'Jack', 'age': 26}
```

```
# update value
```

```
my_dict['age'] = 27
```

```
#Output: {'age': 27, 'name': 'Jack'}
```

```
print(my_dict)
```

```
# add item
```

```
my_dict['address'] = 'Downtown'
```

```
print(my_dict)
```

```
{'name': 'Jack', 'age': 27}  
{'name': 'Jack', 'age': 27, 'address':  
'Downtown'}
```

Q & A

Question
and
Answer



Functions

- Are blocks of organized, reusable code used to perform single or related set of actions
- Provide better modularity and high degree of reusability
- Python supports:
 - Built-in functions like `print()` and
 - User - defined functions

Functions

- **Defining a function:**

- Function blocks starts with a keyword '**def**' followed by **function_name**, parenthesis **(())** and a **colon :**
- Arguments are placed inside these parenthesis
- Function block can have optional statement/comment for documentation as its first line
- Every line inside code block **is indented**
- **return [expression] statement exits the function** by returning an expression to the caller function.
- return statement with no expression is same as return None.

Syntax:

```
def function_name( parameters ):  
    “—optional: Any print statement for documentation”  
    function_suite  
    return [expression]
```

- Parameters exhibit positional behavior, hence should be passed in the same order as in function definition



Functions

- **Calling a Function**

- Defining a function gives it a name, specifies function parameters and structures the blocks of code.
- Functions are invoked by a function call statement/code which may be part of another function

- **Example:**

```
# Defining function print_str(str1)  
def print_str(str1):  
    print("This function prints string passed as an argument")  
    print(str1)  
    return  
  
# Calling user-defined function print_str(str1)  
print_str("Calling the user defined function print_str(str1)")
```

Observe the usage of optional print statement for documentation

Function Call

- **Output:**

```
This function prints string passed as an argument  
Calling the user defined function print_str(str1)
```

Functions

- **Pass arguments to functions:**
 - Arguments are passed by reference in Python
 - Any change made to parameter passed by reference in the called function will reflect in the calling function based on whether **data type of argument passed** is **mutable** or **immutable**
 - In Python
 - **Mutable Data types** include Lists, Sets, Dictionary
 - **Immutable Data types** include Number, Strings, Tuples

Function

Example Program

```
def gcd(a, b):  
    "greatest common divisor"  
    while a != 0:  
        a, b = b%a, a    # parallel assignment  
    return b
```

```
>>> gcd.__doc__  
>>> gcd(12, 20)
```

'greatest common divisor'

4



Functions

- **Scope of variables**

- Determines accessibility of a variable at various portions of the program

- Different types of variables

- **Local variables**

- Variables defined inside the function have local scope
- Can be accessed only inside the function in which it is defined

- **Global variables**

- Variables defined outside the function have global scope
- Variables can be accessed throughout the program by all other functions as well

Example:

```
total = 0

# Function Definition
def add( arg1, arg2 ):
    # Add both the parameters and return total
    total = arg1 + arg2; # total is local variable
    print ("Value of Total(Local Variable): ", total)
    return total;

# Function Invocation
add( 25, 12 );
print("Value of Total(Global Variable): ", total)
```

Output:

```
Value of Total(Local Variable): 37
Value of Total(Global Variable): 0
```

Module

- ❑ Modules refer to a file containing Python statements and definitions.
- ❑ A file containing Python code, for example: **example.py**, is called a module, and its module name would be **example**.
- ❑ Break down large programs into small manageable and organized files.
- ❑ Reusability of code.
- ❑ Most used functions in a module - import it, instead of copying their definitions into different programs.

```
import math  
print("The value of pi is", math.pi)
```

```
import math as m  
print("The value of pi is", m.pi)
```

```
from math import pi, e  
pi  
e
```

Module - Example

```
import re  
emailAddress = input()  
e_var = "(\\w+)@(\\w+)\\.(com)"  
r2 = re.match(e_var ,emailAddress)  
print(r2.group(2))
```

```
helen@gmail.com  
gmail
```

Q & A

Question
and
Answer





Progressio Alas

Thank You