

Programming in Python



Copyright Guideline

© 2016 Infosys Limited, Bangalore, India. All Rights Reserved.

Infosys believes the information in this document is accurate as of its publication date; such information is subject to change without notice. Infosys acknowledges the proprietary rights of other companies to the trademarks, product names and such other intellectual property rights mentioned in this document. Except as expressly permitted, neither this documentation nor any part of it may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, printing, photocopying, recording or otherwise, without the prior permission of Infosys Limited and/ or any named intellectual property rights holders under this document.

Confidential Information

- This Document is confidential to Infosys Limited. This document contains information and data that Infosys considers confidential and proprietary (“Confidential Information”).
- Confidential Information includes, but is not limited to, the following:
 - ❑ Corporate and Infrastructure information about Infosys
 - ❑ Infosys’ project management and quality processes
 - ❑ Project experiences provided included as illustrative case studies
- Any disclosure of Confidential Information to, or use of it by a third party, will be damaging to Infosys.
- Ownership of all Infosys Confidential Information, no matter in what media it resides, remains with Infosys.
- Confidential information in this document shall not be disclosed, duplicated or used – in whole or in part – for any purpose other than reading without specific written permission of an authorized representative of Infosys.
- This document also contains third party confidential and proprietary information. Such third party information has been included by Infosys after receiving due written permissions and authorizations from the party/ies. Such third party confidential and proprietary information shall not be disclosed, duplicated or used – in whole or in part – for any purpose other than reading without specific written permission of an authorized representative of Infosys.

Course Information

Course Code: PY-BASIC

Course Name: Programming in Python

Document Number: 001

Version Number: 1.0

Topics covered in “Programming in Python” course

- Introduction
- Python Basics
- Python Data Variables & Operators
 - Data Variables and its types
 - Operators
 - id() and type() functions
 - Coding Standards
- Control Structures
 - If else
 - elif, Nested if
 - Iteration Control Structures
 - Break, Continue & Pass

Topics covered in “Programming in Python” course

- Data Structures
 - Strings
 - Tuples
 - Lists
 - Sets
 - Dictionary
- Functions
 - Defining & Calling a function
 - Passing Arguments to functions – Mutable & Immutable Data types
 - Different types of arguments
 - Recursion
 - Scope of variables
 - Introduction to Python Tutor

Topics covered in “Programming in Python” course

- Standard Library and Regular Expression
 - Math Module
 - String Module
 - List Module
 - Date & Time Module
- Regular Expression: match, search, replace
- Modules and Package
 - Modules
 - Packages

Topics covered in “Programming in Python” course

- File Operations
 - Open
 - Close
 - Write
 - Read
- Errors and Exception Handling
 - Errors
 - Try....except....else
 - Try....except
 - Try....finally

Introduction



Why Python for beginners?

- Easy – to - learn
 - Code is 3-5 times shorter than Java
 - 5-10 times shorter than C++
- Stepping Stone to Programming universe
 - Python's methodologies can be used in a broad range of applications
- Bridging the Gap between abstract computing and real world applications
 - Python is used as main programming language to do projects using Raspberry Pi
- Rising Demand for Python Programmers
 - Google, Nokia, Disney, Yahoo, IBM use Python
- Open- Source, Object – Oriented, procedural and functional
 - Not only a Scripting language, also supports Web Development and Database Connectivity

Python v/s Java

A simple Program to print "Hello World"

Java Code	Python Code
<pre>public class HelloWorld { public static void main(String args []) { System.out.println ("Hello World!") } }</pre>	<pre>print ("Hello World!")</pre>

Worldwide Python Users

- **Web Development**

- Yahoo Groups, Google, Shopzilla

- **Games**

- Battlefield2, The Temple of Elemental Evil, Vampire

- **Graphics**

- Walt Disney feature Animation, Blender 3D

- **Science**

- National Weather Service
- NASA
- Environmental Systems Research Institute

Evolution of Python

- **Guido Van Rossum** developed Python in **early 1990s** at National Research Institute for Mathematics and Computer Science, Netherlands.
- Named after a circus show Monty Python show.
- Derives its features from many languages like **Java, C++, ABC, C, Modula-3, Smalltalk, Algol-68, Unix shell** and other scripting languages.
- Available under the GNU General Public License (GPL) – Free and open-source software.

Python Versions

- **Python v0.9.0 - February, 1991**

- Features: Exception Handling, Functions and core data types like List, Dictionary, String and others. It was object oriented and had module system

- **Python v1.0 - January 1994**

- Features: Functional Programming tools lambda, map, filter and reduce

- **Python v2.0 - October 2000**

- Features: List comprehensions, Garbage Collector and support for Unicode.

- **Python v3.0 - 2008**

- Known as “Python 3000” and “Py3k”. It is not backward compatible with v2.0 and its other variants. Emphasizes more on removal of duplicate programming constructs and modules

Python Features

- Python is a High - Level, Interpreted, Interactive and Object - Oriented Programming Language
- Features include:
 - Beginners Language
 - Extensive Standard Library
 - Cross Platform Compatibility
 - Interactive Mode
 - Portable and Extendable
 - Databases and GUI Programming
 - Scalable and Dynamic Semantics
 - Automatic Garbage Collection

Configuration

- Download and Install Python 3.5: <https://www.python.org/downloads/>
- Download PyDev_3.8.0 or higher version: <http://www.pydev.org/download.html>
- Download Eclipse Juno or higher version:
<https://www.eclipse.org/downloads/index.php>
- Install Python on your machine

Note: You need eclipse locally installed in your machine

Procedure:

- Copy paste the contents of the plugin folder of PyDev into plugin folder of Eclipse
- In Eclipse open PyDev perspective (Window -> Open Perspective -> Other -> Pydev)
- Create a PyDev Project
- Select Grammar as 3.0
- Configure interpreter by choosing the .exe file of Python installed in your machine

Python Basics



Commenting Style in Python!

Types of Comments:

- A single line comment starts with hash symbol '#' and end as the line ends.
 - These lines are never executed and are ignored by the interpreter.
 - Single → # This is a single line comment
- Multi-line comments starts and ends with triple single quotes ''' or triple "" double quotes
 - Used for documentation
 - Triple → ''' or ""

```
'''
```

Contents here can be used
for documentation

```
'''
```

```
''
```

An example for multi-line
comments with single quotes

```
''
```

Multiline statements

- Python statements always end up with a new line, but it also allows multiline statement using “\” character at the end of line as shown below:

```
result = (8+5)*\  
         2+\  
         9/5
```

- Statements which have (), [], {} brackets and comma, do not need any multiline character to go to next line.

```
customer_details = [101, 'kevin',  
                    '165', 498.24]
```

Print Statement

- Displays the output on the screen of user
- Python has its own String Format Operator as %

```
print('The value of PI is %5.3f' % 3.1417)
print('The value of PI is approximately %5.2f.' % 3.1417)
print( "Latest Python Version is: %d" % 3.5)
print ("%20s : %d" % ('Python', 3000.34 ))
```

Output:

```
The value of PI is 3.142
The value of PI is approximately  3.14.
Latest Python Version is: 3
          Python : 3000
```

User Input in Python

```
name = input("Enter your name")
print("Welcome to session on Programming in Python,", name)
```

Guided Activity: Assignment 1: Using Eclipse IDE to create and execute Python Program

Execute a Python Script

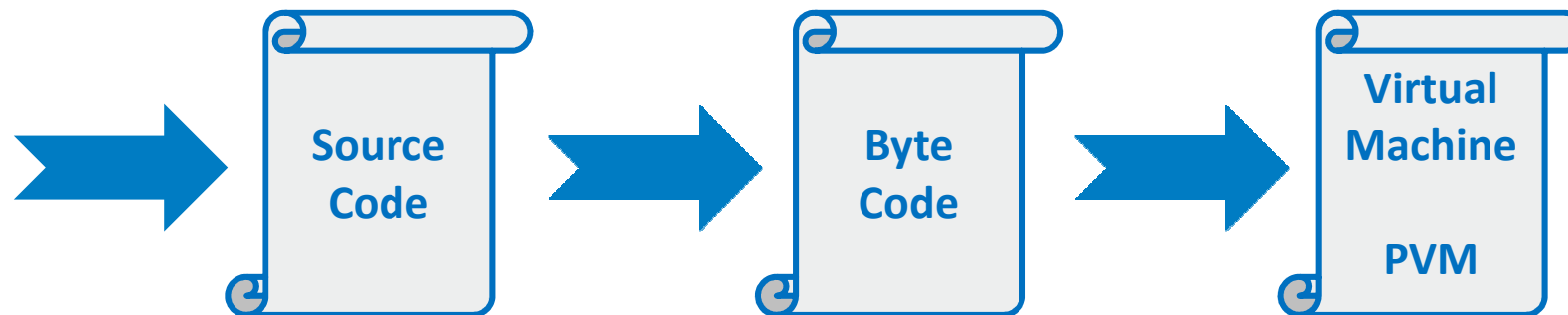
- Execution of python program means execution of the byte code on Python Virtual Machine

```
print("Hello World!")
```

#prints Hello World!

```
print("My first sample python script")
```

#prints My first sample python script



Python Data Variables



Programming Constructs in Python

Guided Activity: Programming constructs in Python - Assignment 2

- Given a real world problem, to solve the problem using a program, we need:
 - Logic
 - High level programming language
 - Programming Fundamentals
 - Identifiers
 - Variables
 - Data types
 - Operators etc

Identifiers

- Are names given to anything that you want to identify in a program
- Helps to refer to that item from any place in the program
- Can start with an underscore (_) or a upper or lower case alphabet
- Can have digits
- Identifiers cannot match any of Python's reserved words
- Are case-sensitive

```
bill_id  
customer_id  
bill_amount
```

**Identify the identifiers
needed to solve the
problem discussed as part
of previous guided activity?**

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	# Integer
customer_name = "John"	# String
bill_amount = 675.45	# Floating-point
x = 5.3 + 0.9j	# complex number
print(customer_id, customer_name, bill_amount)	#prints 101 John 675.45
print(x.real)	#prints 5.3
print(x.imag + 3)	#prints 3.9

Data Types in Python

Category	Data Type	Example
Integer Type	int	675
	long	9669737712
	complex	2 + 5i
Floating Type	float	642.43
Textual	char	C
	String	Infosys
Logical	boolean	True, False

We will now understand different types of operators and how to write a simple program

Guided Activity: Programming constructs in Python - Assignment 3, 4

Python Operators



Operators (1 of 7)

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



Operators (2 of 7)

- **Arithmetic Operators**

- Used for performing arithmetic operations

Operators	Description	Example
+	Additive operator (also used for String concatenation)	$2 + 3 = 5$
-	Subtraction operator	$5 - 3 = 2$
*	Multiplication operator	$5 * 3 = 15$
/	Division operator	$6 / 2 = 3$
%	Modulus operator	$7 \% 2 = 1$
//	Truncation division (also known as floor division)	$10 // 3 = 3$ $10.0 // 3 = 3.0$
**	Exponentiation	$10 ** 3 = 1000$

Operators (3 of 7)

- **Relational Operators**

- Also known as **Comparison operators**
- Used in conditional statements to compare values and take action depending on the result

Operators	Description
==	Equal to
<	Less than
>	Greater than
<=	Lesser than or equal to
>=	Greater than or equal to
!=	Not equal to
<>	Similar to Not equal to

Operators (4 of 7)

- **Assignment Operators**

Operators	Description	Example	Equivalent
=	Assignment from right side operand to left side	c = 50; c = a;	
+=	Add & assigns result to left operand	c += a	c = c + a
-=	Subtract & assigns result to left operand	c -= a	c = c - a
*=	Multiply & assigns result to left operand	c *= a	c = c * a
/=	Divide & assigns result to left operand	c /= a	c = c / a
%=	Calculates remainder & assigns result to left operand	c %= a	c = c % a
//=	Performs floor division & assigns result to left operand	c //= a	c = c // a
**=	Performs exponential calculation & assigns result to left operand	c **= a	c = c ** a

- **Multiple Assignments** – Same value can be assigned to more than one variable

Ex.1: Students Ram, Sham, John belong to semester 6
Ram = Sham = John = 6

Ex.2: a, b, c = 10, 20, 30 is same
as a = 10, b = 20, c = 30

Operators (5 of 7)

- **Bitwise Operators**

- performs bit by bit operation on bits

Operators	Description
&	Binary AND
	Binary OR
^	Binary XOR
~	Binary Ones Complement
<<	Binary Left Shift
>>	Binary Right Shift

Operators (6 of 7)

- **Logical Operators**
 - Are based on Boolean Algebra
 - Returns result as either True or False

Operator	Meaning
and	Short Circuit-AND
or	Short Circuit-OR
not	Unary NOT

Demo: Assignment 5: Programming constructs in Python

Guided Activity: Assignment 6, 7, 8: Programming constructs in Python

Operators (7 of 7)

- **Membership Operators**

- Checks for membership in a sequence of Strings, Lists, Dictionaries or Tuples

Operators	Description
in	Returns to true if it finds a variable in given sequence else false
not in	Returns to true if it does not find a variable in given sequence else false

- **Identity Operators**

- Are used to compare memory locations of 2 objects

Operators	Description
is	Returns to true if variables on either side of operator are referring to same object else false
is not	Returns to false if variables on either side of operator are referring to same object else true

Built-in function: id()

- **id(object)**
 - Returns identity of an object. It is the address of object in memory
 - It will be unique and constant throughout the lifetime of an object

Example:

```
a = 10
b = a
print("Value of a and b before increment")
print("id of a: ",id(a))
print("id of b: ",id(b))
b = a + 1
print("Value of a and b after increment")
print("id of a: ",id(a))
print("id of b: ",id(b))
```

Output

```
Value of a and b before increment
id of a: 1815592664
id of b: 1815592664
Value of a and b after increment
id of a: 1815592664
id of b: 1815592680
```

**Note the change in
address of variable 'b'
after increment**

Built-in function: type()

- Used to identify the type of object

Example:

```
int_a = 10
print("Type of 'int_a':", type(int_a))

str_b = "Hello"
print("Type of 'str_b':", type(str_b))

list_c = []
print("Type of 'list_c':", type(list_c))
```

Output:

```
Type of 'int_a': <class 'int'>
Type of 'str_b': <class 'str'>
Type of 'list_c': <class 'list'>
```

Note: Every variable in Python is a object

Guided Activity: Assignment 9: id() and type() functions - Quiz

Coding Standards in Python

- Set of guidelines
 - To Enhance the readability and Clarity of the program
 - Make it easy to debug and maintain the program
- All the letters in a variable name should be in lowercase
- When there are more than two words in variable name, underscore can be used between internal words
- Use meaningful names for variables
- Limit all lines to a maximum of 79 characters
- A function and class should be separated by 2 blank lines
- Methods within classes should be separated by single blank line
- Always surround binary operators with a space on either side:

Ex: `a = a + 1;`

Coding Standards in Python

Bad Code	Good Code
<pre>a = 10 b = 23 C = 24</pre>	<pre>marks1 = 10 marks2 = 23 marks3 = 24</pre>
<pre>sum = a + b + c</pre>	<pre>sum_of_marks = marks1 + marks2 + marks3</pre>
<pre>avg = sum/3</pre>	<pre>avg_of_marks = sum_of_marks / 3</pre>
<pre>print("Average: ", avg)</pre>	<pre>print("Average: ", avg_of_marks)</pre>

Usage of keyword
'sum' as variable
name will lead to
warning

Guided Activity: Assignment 10: Coding Standards

Control Structures

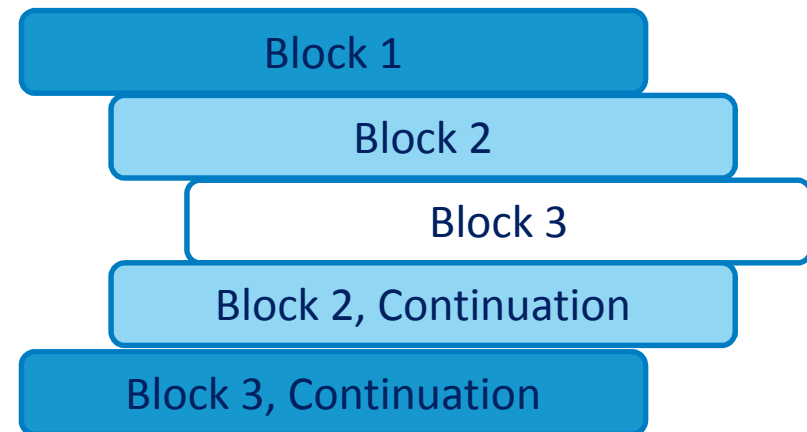


Indentation in Python

- Python uses offside rule notation for coding
- Uses indentation for blocks, instead of curly brackets
- The delimiter followed in Python is a colon (:) and indented spaces or tabs.

Example:

```
x = 3
if x > 5:
    print("true")
    print(x)
else:
    print ("false")
print ("Out of if block")
```



1st level indent → if block

2nd level indent → statements within if or else block

Control Structures

Guided Activity: Assignment 11: Control Structures

- **Decision making statements**

- **if statement:**

- **if** statement checks for a condition and if that is found true a particular set of instructions gets executed

Example:

```
x = 8
if x < 10:
    print("Value of x is %d" %x)
var = 10
if var > 5:
    print ("Hi")           # line belongs to if block
print("I'm out of if")
```

Output:

```
Value of x is 8
Hi
I'm out of if
```

Syntax:

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

**Predict the output of
this code snippet
when value of x = 15?**

Control Structures...

Demo: Assignment 12: Control Structures

- elif statement:
 - **elif** statement is used when there is more than one condition to be checked separately

Example:

```
var=10
if var > 10 :
    print("Hello")
    print(var)
elif var < 10:
    print("Hola in Spanish for Hello")
    print(var)
else:
    print("Hi")
    print(var)
print("End of Program")
```

Syntax:

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

Output:

```
Hi
10
End of Program
```

Note: There is no switch case statement in Python unlike C/C++ language

Guided Activity: Assignment 13, 14, 15, 16: Control Structures

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**

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
```

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
```

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

Guided Activity: Assignment 17: Control Structures

Iterative Statements- break

- Loop Control Statements - break and continue
 - When an external condition is triggered, Exits a loop immediately.
 - **Break Statement:**
 - Terminates the loop statement and transfers execution to the statement immediately following the loop.

Example:

```
var = 3
while var > 0:
    print ("I'm in iteration ",var)
    var -= 1
    if var == 2:
        break
    print ("I'm still in while")
print ("I'm out of while loop")
```

Output:

```
I'm in iteration 3
I'm out of while loop
```

**Observe the output of
this code snippet
when value of var = 5?**

Demo: Assignment 18: break statement

Iterative Statements - continue

– **continue** statement:

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

Example:

```
var = 3
while var > 0:
    print ("I'm in iteration ", var)
    var -= 1
    if var == 2:
        continue
    print ("I'm still in if block")
    print ("I'm still in while")
print ("I'm out of while loop")
```

Output:

```
I'm in iteration 3
I'm in iteration 2
I'm still in while
I'm in iteration 1
I'm still in while
I'm out of while loop
```

Demo: Assignment 19: continue statement

Guided Activity: Assignment 20: Iteration Control Structure - Debugging

Iterative Statements- pass

- **pass** statement:
 - **pass** statement is never executed.
 - Used when a statement is required syntactically but do not want any command or code to execute or if the code need to be implemented in future.
 - Behaves like a placeholder for future code

Example:

```
x = "Joy"
if x == "John":
    print ("Name:",x)
elif x == "Joy":
    pass
else:
    print ("in else")
```

Output:

No Output

Data Structures



Data Structure in Python

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

Strings



Strings

Guided Activity: Assignment 21: Strings

- Accepts 3 types of quotes to assign a string to a variable.
 - single ('), double (") and triple (''' or """)
 - String starts and ends with same type of quote
 - Triple quotes are used to span string across multiple lines.
- Index starts from zero.
- Can be accessed using negative indices. Last character will start with -1 and traverses from right to left.

Syntax:

```
word = 'Programming'  
sentence = "Object Oriented Programming."  
paragraph = """ Python is a Object Oriented Programming Language.  
It is a Beginner's language."""
```

Demo: Assignment 22: Strings

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

Guided Activity: Assignment 23, 24, 25, 26, 27 : Strings

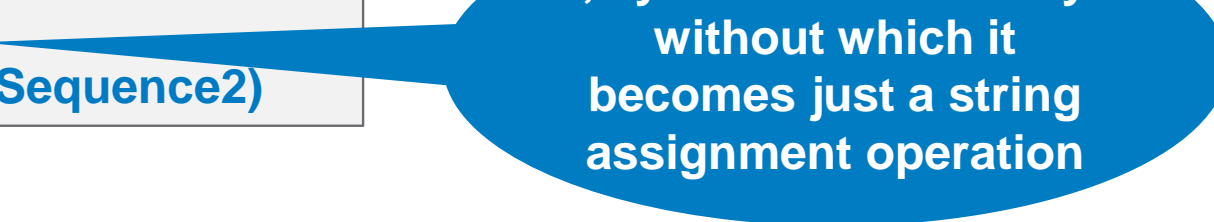
Tuples



Tuples

- An ordered group of sequences separated by symbol , and enclosed inside the parenthesis
- Tuples are **immutable**.

Syntax:

```
tuple1 = () #Creation of empty tuple  
tuple2 = (Sequence1,)   
tuple3 = (Sequence1, Sequence2)
```

, symbol is mandatory
without which it
becomes just a string
assignment operation

Examples:

```
customer = ("John",)  
customers = ('John', 'Joe', 'Jack', 'Jill', 'Harry')
```

Demo: Assignment 28: Operations on Tuples

Guided Activity: Assignment 29: Tuples

Lists



Lists

- An ordered group of sequences enclosed inside square brackets and separated by symbol ,
- Lists are **mutable**.

Syntax:

```
list1 = [] #Creation of empty List  
list2 = [Sequence1,]  
list3 = [Sequence1, Sequence2]
```

In this case
symbol , is NOT
mandatory

Examples:

```
language = ['Python']  
languages = ['Python', 'C', 'C++', 'Java', 'Perl']
```

List Notation and Examples

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

Demo: Assignment 30: Accessing Elements from Lists

Basic List Operations

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

Guided Activity: Assignment 31, 32, 33 : Lists

Sets



Sets

- 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:

<code>set1 = {}</code>	<code>#Creation of empty set</code>
<code>set2 = {"John"}</code>	<code>#Set with an element</code>

Example:

<pre>s1 = set("my name is John and John is my name".split()) s1 = {'is', 'and', 'my', 'name', 'John'}</pre>

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

Demo: Assignment 34: Sets

Operations on Sets

Operation	Equivalent	Operation
len (s)		Length of set 's'
x in s		Membership of 'x' in 's'
x not in s		Membership of 'x' not in 's'
s.issubset(t)	$s \leq t$	Check whether 's' is subset 't'
s.issuperset(t)	$s \geq t$	Check whether 't' is superset of 's'
s.union(t)	$s \cup t$	Union of sets 's' and 't'
s.intersection(t)	$s \cap t$	Intersection of sets 's' and 't'
s.difference(t)	$s - t$	Returns elements in 's' but not in 't'
s.symmetric_difference(t)	$s \Delta t$	Returns elements in either 's' or 't' but not both
s.copy(_)		A new copy of 's'

Guided Activity: Assignment 35: Sets

Dictionary



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
```

Demo: Assignment 36: Dictionary

Guided Activity: Assignment 37: Dictionary

Mutable v/s Immutable Data Types

Mutable Data Type	Immutable Data Type
Sequences can be modified after creation	Sequences cannot be modified after creation
Ex: Lists, Sets, Dictionary	Ex: Strings, Tuples
Operations like add, delete and update can be performed	Operations like add, delete and update cannot be performed

Example

```
int_list = [12, 14, 9]
int_tuple = (12, 14, 19)

int_list[0] = 7
#prints [7, 14, 9]
print("List: ", int_list)

int_tuple[0] = 7
#Prints TypeError: 'tuple' object does not support
#item assignment
print("Tuple: ", int_tuple)
```

Output

```
List: [7, 14, 9]
int_tuple[0] = 7
TypeError: 'tuple' object does
not support item assignment
```

Functions



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 (Cont...)

- **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 (Cont...)

- **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 (Cont...)

- **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

Functions (Cont...)

- Pass arguments to functions: Immutable Data Type - Number

Example:

#Function Definition

```
def change(cust_id):  
    cust_id += 1  
    print("Customer Id in function definition: ", cust_id)  
    return
```

Function Invocation with arguments of immutable data type

```
cust_id = 100  
print("Customer Id before function invocation: ", cust_id)  
change(cust_id)  
print("Customer Id after function invocation: ", cust_id)
```

Output:

```
Customer Id before function invocation: 100  
Customer Id in function definition: 101  
Customer Id after function invocation: 100
```

Observe that customer id remains unchanged even after function invocation

Functions (Cont...)

- Pass arguments to functions: Mutable Data Type - List

Example:

#Function Definition

```
def change(list_cust_id):
```

#Assign new values inside the function

```
    list_cust_id.append([10, 20, 30])
```

```
    print("Customer Id in function definition: ", list_cust_id)
```

```
    return
```

Function Invocation with arguments of immutable data type

```
list_cust_id = [100, 101, 102]
```

```
print("List of Customer Id before function invocation: ", list_cust_id)
```

```
change(list_cust_id)
```

```
print("Customer Id after function invocation: ", list_cust_id)
```

Output:

```
List of Customer Id before function invocation: [100, 101, 102]
```

```
List of Customer Id in function definition: [100, 101, 102, [10, 20, 30]]
```

```
List of Customer Id after function invocation: [100, 101, 102, [10, 20, 30]]
```

Functions (Cont...)

- Different types of formal arguments:
 - Required arguments
 - Keyword arguments
 - Default arguments
 - Variable – length arguments

Demo: Assignment 38: Functions – Pass by Reference

Guided Activity: Assignment 39: Functions – Pass by Reference

Functions (Cont...)

- **Required arguments**

- Arguments follow positional order
- No. of arguments and the order of arguments in the function call should be exactly same as that in function definition

Example:

Function Definition

```
def print_str(str1):  
    print("This function prints the string passed as an argument")  
    print(str1)  
    return
```

Function Invocation without required arguments

```
print_str()
```

Output:

```
TypeError: print_str() missing 1 required positional argument: 'str1'
```

Functions (Cont...)

- **Keyword arguments**

- when used in function call, the calling function identifies the argument by parameter name
- Allows you to skip arguments or place them out of order
- Python Interpreter uses the keyword provided to match the values with parameters

Example:

Function Definition

```
def customer_details (cust_id, cust_name):  
    print("This function prints Customer details")  
    print("Customer Id: ",cust_id)  
    print("Customer Name: ",cust_name)  
    return
```

Function Invocation with Keyword arguments

```
customer_details(cust_name = "John", cust_id = 101)
```

Output:

```
This function prints Customer details  
Customer Id: 101  
Customer Name: John
```

**Observe the change in
positional order of
arguments**

Functions (Cont...)

- **Default Arguments:**

- Assumes a default value if the value is not specified for that argument in the function call

Example:

Function Definition

```
def customer_details (cust_name, cust_age = 30):  
    print("This function prints Customer details")  
    print("Customer Name: ",cust_name)  
    print("Customer Age: ",cust_age)  
    return
```

Function Invocation with Default arguments

```
customer_details(cust_age = 25, cust_name = "John")  
customer_details(cust_name = "John")
```

Output:

```
This function prints Customer details  
Customer Name: John  
Customer Age: 25  
This function prints Customer details  
Customer Name: John  
Customer Age: 30
```

Observe the usage of
default value for cust_age
argument

Functions (Cont...)

- Variable-length arguments

- Used to execute functions with more arguments than specified during function definition
- unlike required and default arguments, variable arguments are not named while defining a function

Syntax:

```
def functionname([formal_args,] *var_args_tuple ):
    “—optional: Any print statement for documentation”
    function_suite
    return [expression]
```

- An asterisk ‘*’ is placed before variable name to hold all non-keyword variable arguments
- ***var_args_tuple** is empty if no additional arguments are specified during function call

Functions (Cont...)

- Variable-length arguments

Example:

Function Definition

```
def customer_details (cust_name, *var_tuple):  
    print("This function prints Customer Names")  
    print("Customer Name: ",cust_name)  
    for var in var_tuple:  
        print(var)  
    return
```

Function Invocation with Variable length arguments

```
customer_details("John", "Joy", "Jim", "Harry")  
customer_details("Mary")
```

Output:

```
This function prints Customer Names  
Customer Name: John  
Joy  
Jim  
Harry  
This function prints Customer Names  
Customer Name: Mary
```

Invoke this
function without
arguments and
observe the
output

Functions (Cont...)

- **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
```


Usage of keyword 'Global'

- Used to access the variable outside the function

Example:

```
total = 0
```

Function Definition

```
def add( arg1, arg2 ):
```

```
    # Add both the parameters and return total
```

```
    global total
```

```
    total = arg1 + arg2; # Here total is made global variable
```

```
    print ("Value of Total(inside the function): ", total)
```

```
    return total;
```

Function Invocation

```
add( 25, 12 );
```

```
print("Value of Total(outside the function): ", total)
```

Observe the usage
of keyword 'global'

Output:

```
Value of Total(inside the function): 37
```

```
Value of Total(outside the function): 37
```

Variable total is
accessible outside
the function

Recursion

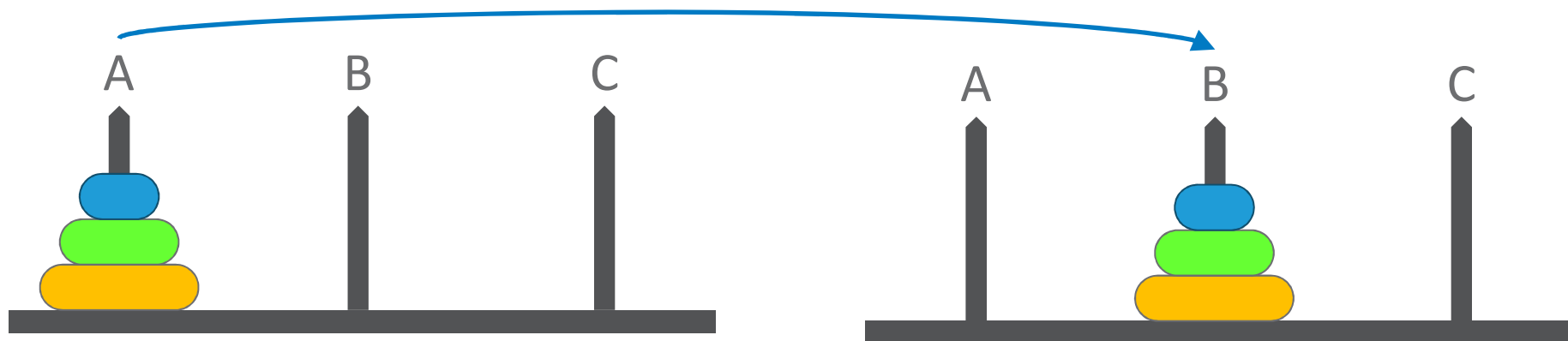
- A method invoking itself is referred to as Recursion
- Typically, when a program employs recursion the function invokes itself with a smaller argument
- Computing factorial(5) involves computing factorial(4), computing factorial(4) involves computing factorial(3) and so on
- Often results in compact representation of certain types of logic and is used as substitute for iteration

Demo: Assignment 40a: Recursion

Guided Activity: Assignment 40b, 41, 42: Recursion

Towers of Hanoi problem

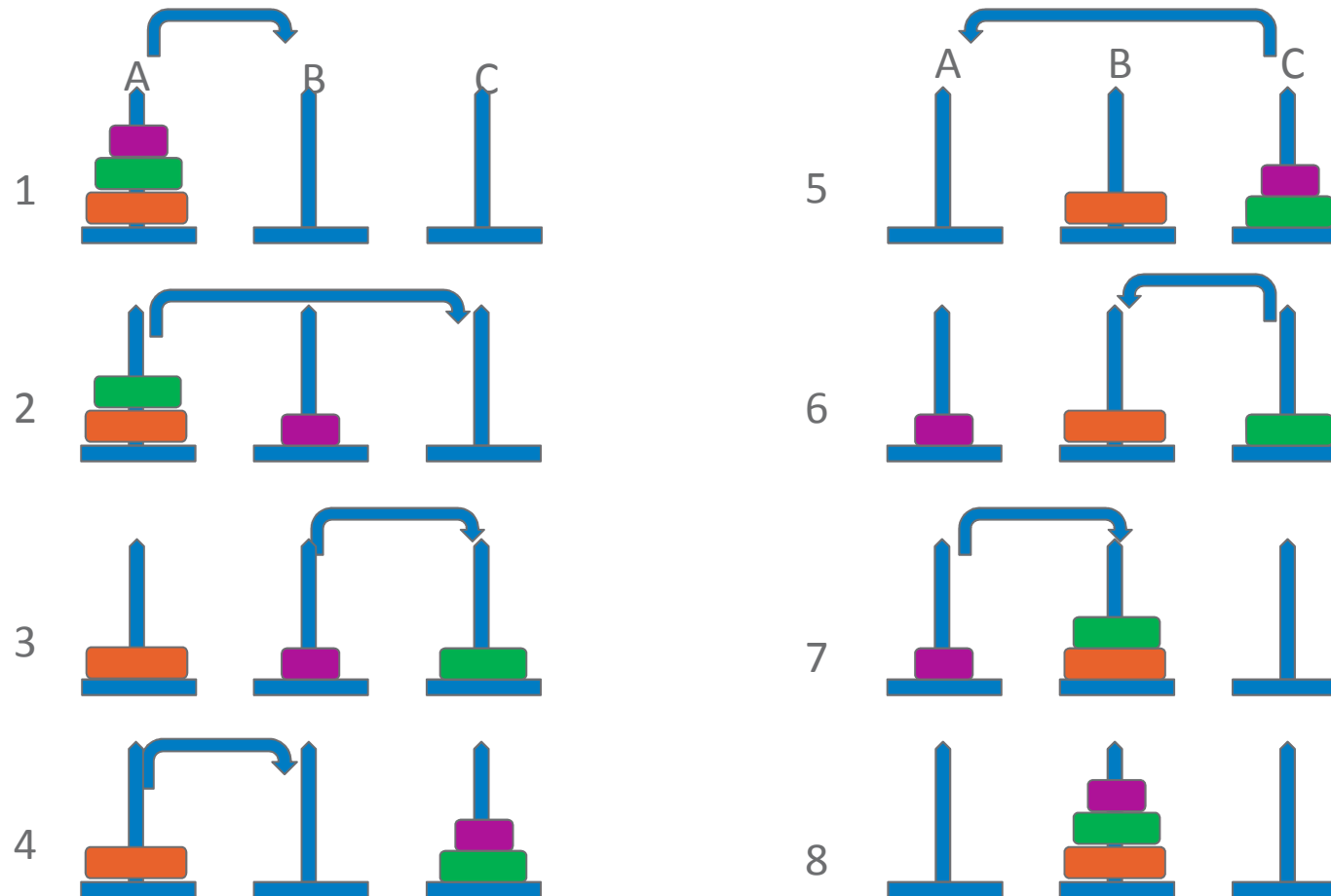
- Problem was discovered by the French mathematician Edouard Lucas in 1883.
- Used to learn recursion
- Rules:
 - Move only one disk at a time. Rings must be in decreasing size
 - No move should result in a larger disk on top of a smaller disk
 - For temporarily holding a disk, the third tower can be used



Transfer the 3 disks from tower A to tower B

Towers of Hanoi - 3 discs

Solution:



Introduction to Python Tutor

- A visualization tool Created by Philip Guo
- Helps in understanding step-by-step execution of each line of a program's source code.
- It can be used to write and visualize programs in Python, Java, JavaScript, Ruby, C and C++
- Click on the link below to visualize your code using Python Tutor and follow the steps in following 3 slides

<http://www.pythontutor.com/>

- Suggested Sections of examples on Python Tutor:
 - Basic
 - Intro – a 10min introduction to Python Programming Fundamentals
 - Math
 - Objects
 - Linked Lists

Introduction to Python Tutor...

Python Tutor - Visualize Python Programs

www.pythontutor.com

VISUALIZE [Python](#), [Java](#), [JavaScript](#), [TypeScript](#), [Ruby](#), [C](#), and [C++](#) programs

Python Tutor, created by [Philip Guo](#), helps people overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code.

Using this tool, you can write [Python](#), [Java](#), [JavaScript](#), [TypeScript](#), [Ruby](#), [C](#), and [C++](#) programs in your Web browser and visualize what the computer is doing step-by-step as it executes those programs. So far, over **1.5 million people in over 180 countries** have used Python Tutor to visualize over 13 million pieces of code, often as a supplement to textbooks, lecture notes, and online programming tutorials.

[Start writing and visualizing code now!](#)

For example, here is a visualization showing a Python program that [recursively](#) finds the sum of a list of numbers.

```
Python 2.7
```

```
→ 1 def listSum(numbers):
2     if not numbers:
3         return 0
4     else:
5         (f, rest) = numbers
6         return f + listSum(rest)
7
8 myList = (1, (2, (3, None)))
9 total = listSum(myList)
```

[Edit code](#)

Frames Objects

Click here to start coding and visualizing

Introduction to Python Tutor...

The screenshot shows the Python Tutor web application. At the top, there's a navigation bar with a logo and the text 'Visualize Python, Java, JavaScript, TypeScript, Ruby, C, and C++ code execution'. Below this, there's a 'Start shared session' button and a link 'What are shared sessions?'. The main area has a title 'Python Tutor: Visualize Python, Java, JavaScript, TypeScript, Ruby, C, and C++ code execution'. Below the title, there's a dropdown menu labeled 'Write code in' with 'Python 3.3' selected. To the right of this dropdown, a callout box says 'Select programming language Python 3.3 from drop-down menu'. Below the dropdown is a code editor with the following code:

```
1 total = 0
2 # Function Definition
3 def add( arg1, arg2 ):
4     # Add both the parameters and return total
5     total = arg1 + arg2; # Here total is local variable
6     print ("Value of Total(Local Variable): ", total)
7     return total;
8 # Function Invocation
9 add( 25, 12 );
10 print("Value of Total(Global Variable): ", total)
```

To the left of the code editor, a callout box says 'Write your code here'. Below the code editor, there's a text prompt: 'Please support our research and keep this tool free by [filling out this short survey](#). If you are at least 60 years old, please also [fill out this survey](#).' Below this is a 'Visualize Execution' button. To the right of this button, a callout box says 'Click on Visualize execution to visualize your code'. At the bottom, there are three dropdown menus: 'hide exited frames [default]', 'inline primitives & nested objects [default]', and 'draw pointers as arrows [default]'.

Introduction to Python Tutor...

The screenshot displays the Python Tutor interface. The main code area shows a Python 3.3 script with a function definition and an invocation. A red circle highlights the execution step at line 8, with a legend below indicating that a green arrow represents the line just executed and a red arrow represents the next line to execute. The program output shows the result of the function call. To the right, the 'Frames' and 'Objects' panels show the state of the program. The 'Global frame' contains variables 'total' and 'add'. The 'add' function frame shows its arguments and the return value. A red arrow points from the 'add' function in the 'Global frame' to the 'add' function in the 'Objects' panel. A red circle highlights the 'Forward >' button in the navigation controls.

Visualize Python, Java, JavaScript, TypeScript, Ruby, C, and C++ code execution

Python 3.3

```
1 total = 0
2 # Function Definition
3 def add( arg1, arg2 ):
4     # Add both the parameters and return total
5     total = arg1 + arg2; # Here total is local variable
6     print ("Value of Total(Local Variable): ", total)
7     return total;
8 # Function Invocation
9 add( 25, 12 );
10 print("Value of Total(Global Variable): ", total)
```

Edit code

<< First < Back Step 8 of 9 Forward > Last >>

Program output:
Value of Total(Local Variable): 37

Frames

Global frame
total
add

Objects

function
add(arg1, arg2)

add

arg1	25
arg2	12
total	37
Return value	37

Observe the change in values of variables during execution and contents of frames and objects

Click here to visualize the step-by-step execution of your code

Standard Library



Math Module

- Provides access to mathematical functions like power, logarithmic, trigonometric, hyperbolic, angular conversion, constants etc;
- Few functions are described below:

Function	Description
abs(x)	Absolute value of x: the (positive) distance between x and zero
ceil(x)	Ceiling of x: smallest integer not less than x
cmp(x, y)	-1 if $x < y$, 0 if $x == y$, or 1 if $x > y$
exp(x)	Exponential of x: e^x
floor(x)	Floor of x: the largest integer not greater than x
max(x1, x2,...)	Largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	Smallest of its arguments: the value closest to negative infinity
pow(x, y)	Value of x^y
round(x [,n])	x rounded to n digits from the decimal point.
sqrt(x)	Square root of x for $x > 0$

String Module

- Includes built-in methods to manipulate strings. Consider the string, str = Infosys

Method	Result	Description
str.count("s")	Returns count of occurrence of character "s" in string str	2
str.startswith("s")	Returns true if string str starts with character "s"	false
str.endswith("s")	Returns true if string str ends with character "s"	true
str.find("s")	Returns index position of character "s" in string str if found else -1	4
str.replace("s", "S")	Replaces all occurrences of character "s" with character "S" in string str	InfoSyS
str.isdigit()	Checks if all the characters in string str are digits and returns true or false accordingly	false
str.upper()	Converts all the characters in string str to uppercase	INFOSYS
str.lower()	Converts all the characters in string str to lowercase	infosys

Guided Activity: Assignment 43: Strings built-in functions

List module

- Built-in functions and methods in lists

Function	Description
<code>cmp(list1, list2)</code>	Compares elements of both lists
<code>len(list)</code>	Gives total length of list
<code>max(list)</code>	Returns item from the list with maximum value
<code>min(list)</code>	Returns item from the list with minimum value
<code>list(seq)</code>	Converts a tuple to list
<code>list.append(obj)</code>	Appends object obj to list
<code>list.count(obj)</code>	Returns count of how many times obj occurs in list
<code>list.insert(index, obj)</code>	Inserts object obj into list at offset index
<code>obj = list.pop()</code>	Removes the item at position -1 from list and assigns it to obj
<code>list.remove(obj)</code>	Removes object obj from list
<code>list.reverse()</code>	Reverses the order of items in list
<code>sorted(list)</code>	Sorts items in list

Guided Activity: Assignment 44, 45, 46, 47, 48: Lists

Date and Time Module

- Supplies classes for manipulating dates and times in both simple and complex ways.
- Import time module. Ex: `Print(time.localtime())`

Function	Description
<code>time.clock()</code>	Returns current time in seconds, given as a floating point number
<code>time.gmtime()</code>	Returns current UTC date and time (not affected by timezone)
<code>time.localtime()</code>	Returns time based on the current locality (is affected by timezone)
<code>time.timezone()</code>	Returns the number of hours difference between your timezone and the UTC time zone (London)
<code>time.time()</code>	Returns the number of seconds since January 1 st 1970.
<code>time.sleep(secs)</code>	Suspends execution of the current thread for the given number of seconds
<code>time.daylight()</code>	Returns 0 if you are not currently in Daylight Savings Time

Regular Expressions in Python



Regular Expression

- Special sequence of characters to match or find other strings or set of strings, using a specialized syntax held in a pattern.
- Widely used in languages like UNIX, PHP, Perl.
- Module **re** provides support for regular expressions in Python.
- **re** module raises `re.error` exception if an error occurs while compiling or using a regular expression

Functions in re module

- Import **re** module to access methods in re module
- Some important functions in **re** module:
- **match** function
 - attempts to match re pattern to string with optional flags

Syntax: `re.match(pattern, string, flags=0)`

- **findall** function
 - Returns all non – overlapping matches of re pattern in a given string

Syntax: `re.findall(pattern, string, flags=0)`

Guided Activity: Assignment 49, 50: Regular Expression

Functions in re module

- **search** function
 - Searches for first occurrence of re pattern within string with optional flags.

Syntax: `re.search(pattern, string, flags=0)`

- **match** checks for a match only at the beginning of string, while **search** checks for a match anywhere in the string.
- **sub** function
 - Substitutes all occurrences of the re pattern in string with repl, replacing all occurrences unless max provided. Returns modified string.

Syntax: `re.sub(pattern, repl, string, max=0)`

Guided Activity: Assignment 51, 52: Regular Expression

Module



Module

- Allows logical organization of code.
- Grouping related code into module makes it easier to understand and use.
- Can be used to define functions, classes and variables.
- It is a file and may have runnable code.
- ***import*** Statement
 - Any Python source file can be used as a module by executing an import statement in any other Python source file

Syntax:

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

- When the interpreter encounters the import statement, it imports the module if it is present
- Module is loaded only once regardless of the number of times it is imported

Packages and its elements



Packages

- Packages are namespaces which contain multiple packages and modules
- Collection of modules in directory
- Must have `__init__.py` file
- May contain subpackages
- `__init__.py` can be empty or it contains valid python code
- `__init__.py` indicates that the directory it contains is a package and it can be imported the same way as a module
 - **Ex: `foo.abc`**
 - Module `abc` belongs to package named `foo`
 - Users of the package can import individual modules from the package
 - **Ex: `import sound.effects.echo`**
 - This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

File Operations



File operations

- A file is a chunk of logically related data or information which can be used by computer programs.
- Python provides some basic functions to manipulate files
- **open** Function

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

- **close** Function

Syntax: `fileObject.close()`

- **write** Function

Syntax: `fileObject.write(string)`

- **read** Function

Syntax: `fileObject.read([count])`

open Function

- Used to open or create a file

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

- **file_name:** Name of the file that you want to access
 - **access_mode:** determines the mode in which the file is to be opened, like read, write, append, etc
 - **buffering:** 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.
- Once a file is opened following list of attributes can get information related to file

Attribute	Returns (Description)
file.close	true if file is closed, false otherwise.
file.mode	access mode with which file was opened
file.name	name of the file.

open Function...

Example:

Open a file

```
result = open("foo.txt", "w")  
print("Name of the file: ", result.name)  
print("Closed or not : ", result.closed)  
print("Opening mode : ", result.mode)
```

Output:

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

`close()` Function

- flushes any unwritten information and closes the file object, after which no more writing can be done.

Syntax:

```
fileObject.close()
```

Example:

```
# Open a file
result = open("foo.txt", "w")
print("Name of the file: ", result.name)

# Close opened file
result.close()
```

Output:

```
Name of the file: foo.txt
```

write Function

- Writes any string to an open file.
- Strings can have binary data or text data.
- Does not add a newline character ('\n') to the end of the string

Example:

Syntax:

fileObject.write(string)

Open a file

```
result = open("foo.txt", "w")  
result.write( "Python is a great language.\nYeah its great!!\n");
```

Close opened file

```
result.close()
```

Output:

```
Contents of file foo.txt:  
Python is a great language.  
Yeah its great!!
```

read Function

- Reads a string from an open file.

Syntax: `fileObject.read([count])`

- Parameter 'count' is the number of bytes to be read from the opened file.
- It starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

Example:

```
# Open a file
result = open("foo.txt", "r+")
sentence = result.read(10);
print ("Read String is : ", sentence)

# Close opened file
result.close()
```

Output:

```
Read String is : Python is
```

Errors and Exception Handling



Errors and Exception handling

- Used to handle any unexpected error in Python programs
- Few Standard Exceptions:

Exception Name	Description
Exception	Base class for all exceptions
Arithmetic Error	Base class for all errors that occur for numeric calculation
Floating Point Error	Raised when a floating point calculation fails.
Zero Division Error	Raised when division or modulo by zero takes place for all numeric types.
IO Error	Raised when an input / output operation fails, such as print() or open() functions when trying to open a file that does not exist.
Syntax error	Raised when there is a error on Python syntax
Indentation error	Raised when indentation is not specified properly
Value Error	Raised when built-in-function for a data type has a valid type of arguments, but the arguments have invalid values specified
Runtime Error	Raised when a generated error does not fall into any category

Handling an Exception

- Exception is an event, which occurs during the execution of program and disrupts the normal flow of program's instructions.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a ***try:*** block.
- After the ***try:*** block, include an ***except:*** statement, followed by a block of code which handles the problem as elegantly as possible.
- Different ways of Exception Handling in Python are:
 - ***try....except...else***
 - ***try...except***
 - ***try...finally***

Handling an Exception...

- ***try...except...else***

- A single try statement can have multiple except statements
- Useful when we have a try block that may throw different types of exceptions
- Code in else-block executes if the code in the try: block does not raise an exception

Syntax:

try:

You do your operations here;

.....

except ExceptionA:

If there is ExceptionA, then execute this block.

except ExceptionB:

If there is ExceptionB, then execute this block.

.....

else:

If there is no exception then execute this block

Example:

try:

fh = open("testfile", "w")

fh.write("This is my test file")

except IOError:

print ("Error: can't find file or read data")

else:

print ("Written content to file successfully")

fh.close()

Output:

Written content to file successfully

Try to open the same file when you do not have write permission, it raises an exception

Handling an Exception...

- ***try...except..***
 - Catches all exceptions that occur
 - It is not considered as good programming practice though it catches all exceptions as it does help the programmer in identifying the root cause of the problem that may occur.

Syntax:

try:

You do your operations here;

.....

except ExceptionA:

If there is ExceptionA, then execute this block.

except ExceptionB:

If there is ExceptionB, then execute this block.

.....

Example:

try:

fh = open("testfile", "w")

fh.write("This is my test file for exception handling!!")

except IOError:

print ("Error: can't find file or write data")

Output:

Error: can't find file or write data

Try to write to the file when you do not have write permission, it raises an exception

Handling an Exception...

- *try...finally..*
 - finally block is a place to put any code that must execute irrespective of try-block raised an exception or not.
 - else block can be used with finally block

Syntax:

try:

You do your operations here;

.....

Due to any exception, this may be skipped.

finally:

This would always be executed.

.....

Example:

try:

fh = open("testfile", "w")

fh.write("This is my test file for exception handling!!")

finally:

print("Error: can't find file or write data")

Output:

Error: can't find file or write data

Try to write to the file when you do not have write permission

References

Books:

- Head First Programming, Apress Publications
- Head First Python, Apress Publications
- Beginning Python, Apress Publications

E-Books:

- <http://www.diveintopython3.net/>

Tutorials:

- <http://www.pythontutor.com/>
- <http://www.learnpython.org/>
- <https://docs.python.org/2/tutorial/index.html>
- http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_3/Intro
<https://developers.google.com/edu/python/introduction>

Thank You



© 2013 Infosys Limited, Bangalore, India. All Rights Reserved. Infosys believes the information in this document is accurate as of its publication date; such information is subject to change without notice. Infosys acknowledges the proprietary rights of other companies to the trademarks, product names and such other intellectual property rights mentioned in this document. Except as expressly permitted, neither this documentation nor any part of it may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, printing, photocopying, recording or otherwise, without the prior permission of Infosys Limited and/or any named intellectual property rights holders under this document.

Infosys® | Building
Tomorrow's Enterprise

Appendix



Python v/s Java

Python	Java
Dynamic Typing and weak typing	Static Typing and strong typing
Does not require declaration of variables before usage	Requires declaration of variables before usage
Allows change of data type of a variable during run-time	Does not allow change of data type of a variable after declaration
Uses indentation to separate code into blocks	Uses brackets to separate code into blocks
Easy to learn	Needs more time compared to Python
Simple and consistent syntax	Complex syntax
Helps beginners to concentrate on problem decomposition and data type design	Beginners spend their initial days on learning to declare and use data variables
Concise, compact and clean looking syntax	Not compact
Print statement automatically inserts spaces and newline	'\n' or a space should be used to insert a space or newline

Type Conversion

- Conversion of data type of a variable from one to other
- Following are few built-in types available to convert between types

Function	Description
<code>int(a)</code>	Converts 'a' to an integer
<code>long(a)</code>	Converts 'a' to a long integer
<code>float(a)</code>	Converts 'a' to a floating – point number
<code>complex (real [,imag])</code>	Creates a complex number
<code>str(a)</code>	Converts object 'a' to a string representation
<code>eval(a)</code>	Evaluates a string and returns an object
<code>tuple(a)</code>	Converts 'a' to a tuple
<code>list(a)</code>	Converts 'a' to a list.
<code>set(a)</code>	Converts 'a' to a set.
<code>dict(a)</code>	Creates a dictionary. 'a' must be a sequence of (key, value) tuples.

Lambda or Anonymous Functions

- '*lambda*' keyword is used to define anonymous functions
- Anonymous functions are not defined in the standard manner using '*def*' keyword
- Can take any number of arguments but, return only one value in the form of an expression
- Cannot be direct call to print as lambda requires an expression
- They have their own namespace and cannot access variables in either in local or global namespaces
- **Note:** They are not equivalent to inline statements in C and C++
- Syntax contains only a single statement

Syntax:

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


Lambda or Anonymous Functions

Example :

Function definition

```
total = lambda num1, num2: num1 + num2
```

Function Invocation

```
print ("Value of total : ", total( 8, 25 ))
```

```
print ("Value of total : ", total( 56, 15 ))
```

Output:

```
Value of total : 33
```

```
Value of total : 71
```

Multithreading

- Is running several threads concurrently similar to running different programs concurrently
- Thread
 - Benefits:
 - Threads within a process can share same data space
 - Can share information or communicate more easily than if they were in separate processes
 - Called light-weight process and are cheaper than processes with less memory over-head
- Has a beginning, an execution sequence and conclusion
- Has an instruction pointer to keep track of the context it is currently running
- Python provides built-in functions to perform operations on threads
- Threads can be interrupted or temporarily put on hold (known as sleeping) while others are running – called yielding