# Python

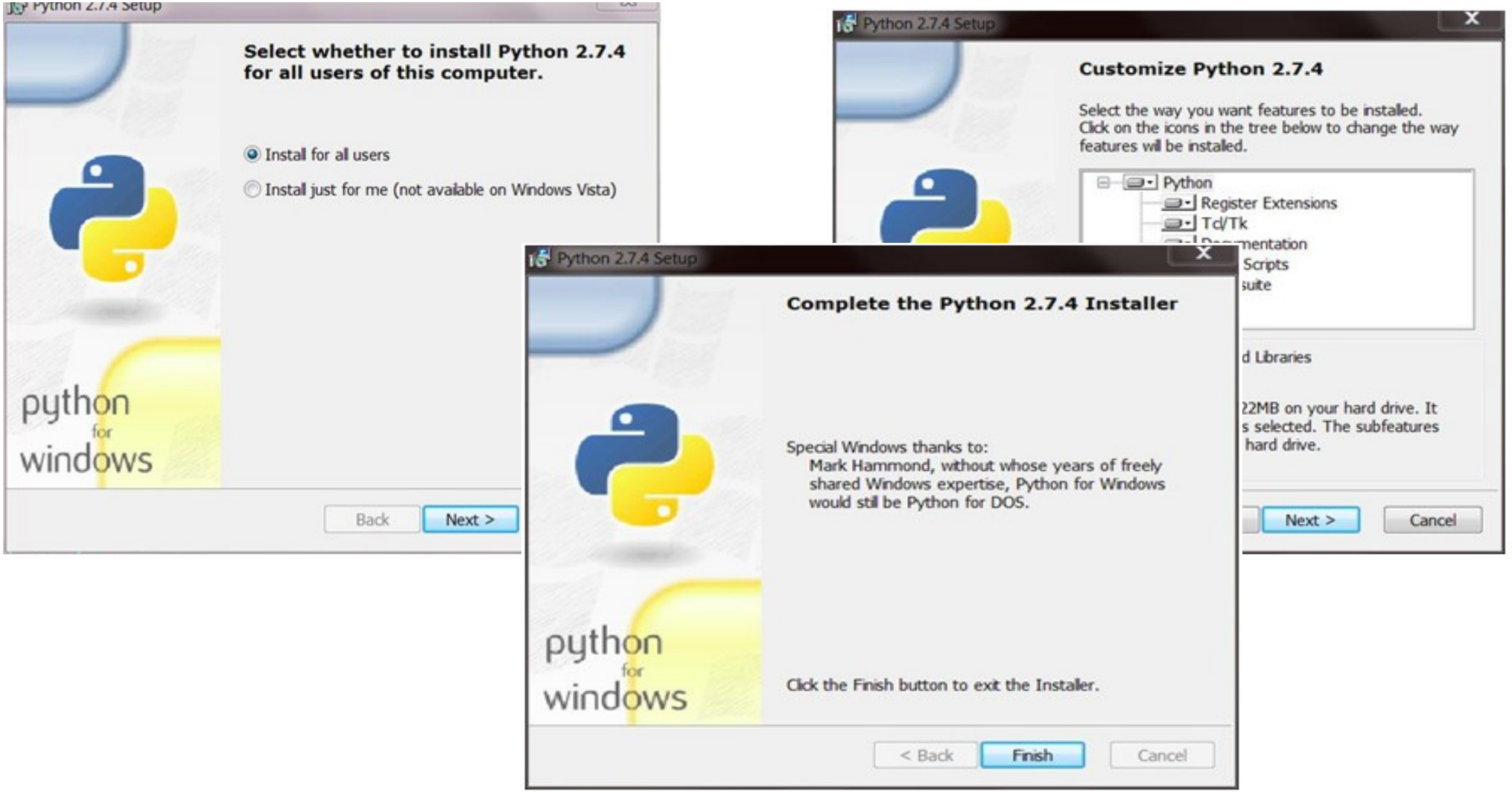
- Python is an programming language created by Guido Rossum in 1989.
- Python is Interpreted language.
- Python is Dynamically typed.
- Python is a Beginner's Language
- Write less code and do more.
- Many large companies use the Python programming language include NASA, Google

# Features

- Easy-to-learn, read and maintain
- A broad standard library
- Interactive Mode
- Portable and extensible
- Databases

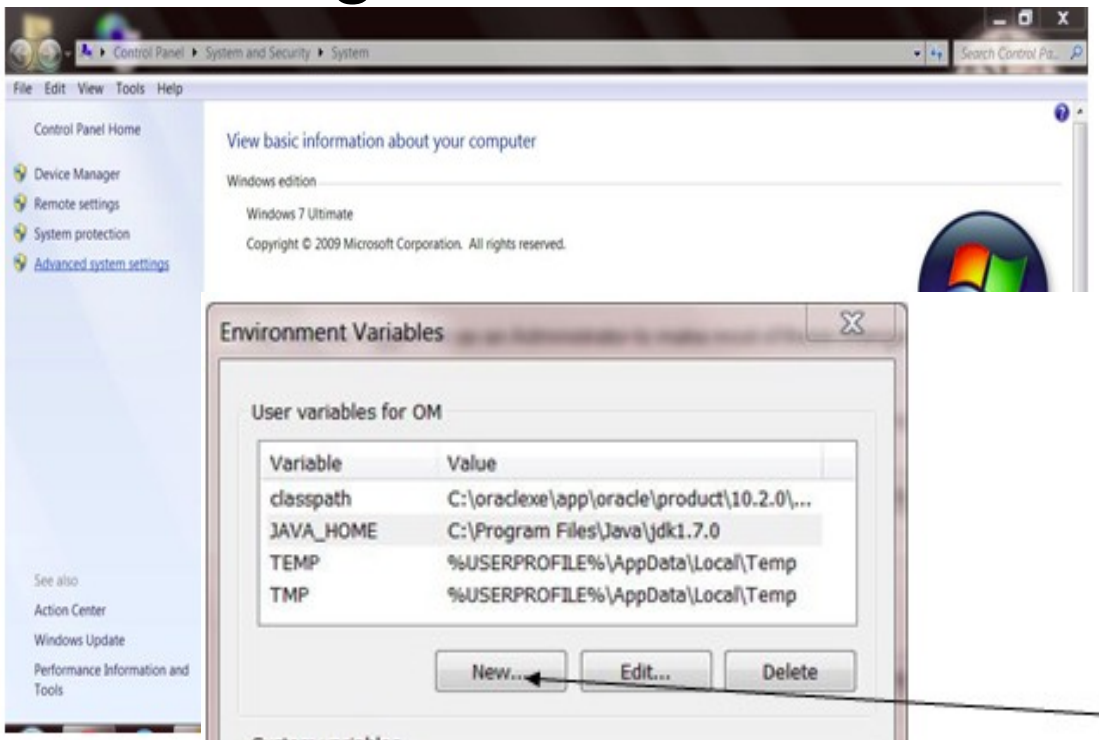
# Installing Python

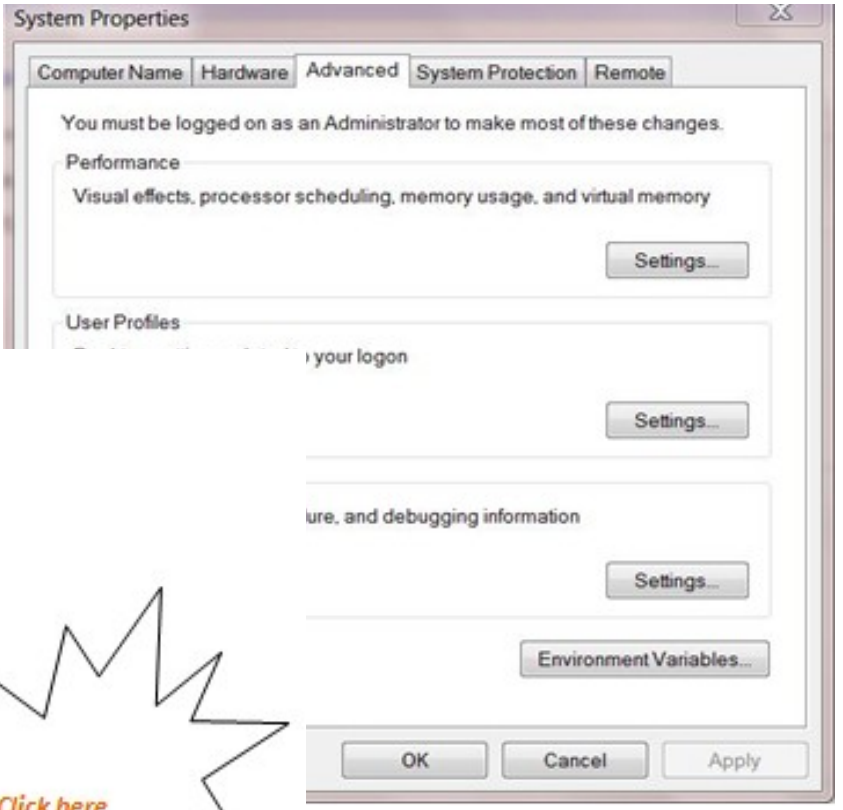
Download Link: https://www.python.org/downloads/



# Installing Python

- Setting Path



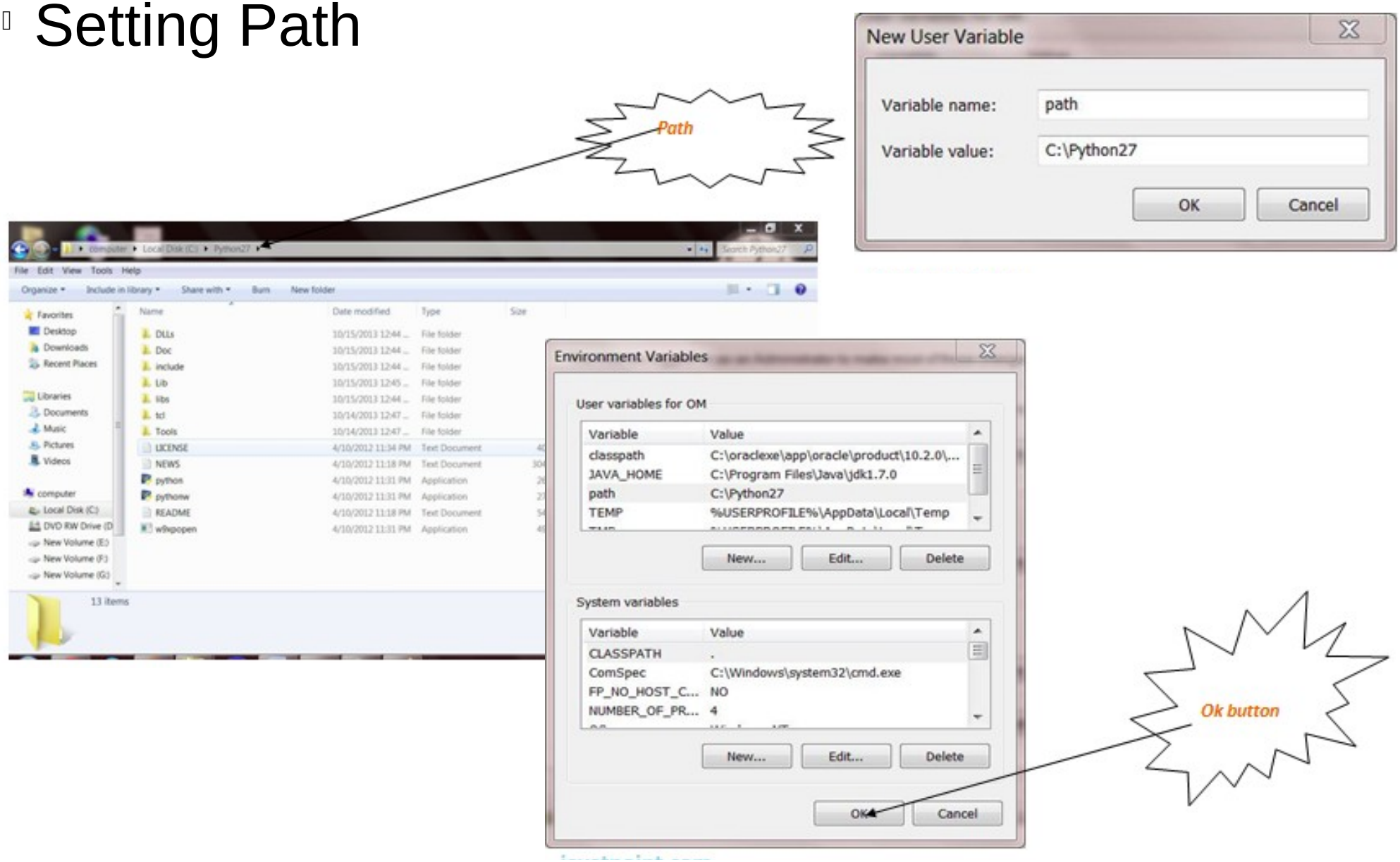







# Installing Python

- Setting Path



# Installing Python

- Ubuntu
  - Type python
- Installing Text Editor
  - Notepad++ (https://notepad-plus-plus.org/)
  - Sublime editor (http://www.sublimetext.com/)
- Compiling a python
  - Python <programname.py>

# Basic Syntax

- Interactive Mode Programming
- Script Mode Programming
- Python Identifiers
  - It is a name of a variable
  - An identifier starts with a letter A to Z or a to z or an (_) followed by zero or more letters, underscores and digits (0 to 9)
  - Python does not allow punctuation characters such as @, $, and % within identifiers.
  - Python is a case sensitive programming language. Thus, Python and python are two different identifiers in Python.

# Basic Syntax

- Reserved Words
  and,  exec, not, assert, finally, or, break, for, pass, class, from, print, continue, global, raise, def, if, return, del, import, try, elif, in, while, else, is, with, except, lambda, yield
- Lines and Indentation
  - **if True:**
  - **  print "True"**
  - **else:**
  - **  print "False"**

- However, the following block generates an error −
  - **if True:**
  - **  print "Answer"**
  - **  print "True"**
  - **else:**
  - **  print "Answer"**
  - **  print "False"**

# Basic Syntax

- Multi-Line Statements
  - **total = item_one + **
  - **item_two + **
  - **item_three**
- Quotation in Python
  - **word = 'word'**
  - **sentence = "This is a sentence."**
  - **paragraph = """This is a paragraph. It is**
  - **made up of multiple lines and sentences."""**
- Comments in Python
  - **# First comment**
  - **print "Hello, Python!" # second comment**
  - **name = "Madisetti" # This is again comment**

# Variable Types

- **Def**: Variables are nothing but reserved memory locations to store values.
- Assigning Values to Variables
  - Python variables do not need explicit declaration
  - The equal sign (=) is used to assign values to variables.
- Code:
  - **counter = 100 # An integer assignment**
  - **miles = 1000.0 # A floating point**
  - **name = "John" # A string**

  - **print counter**
  - **print miles**
  - **print name**

# Variable Types

- Multiple Assignment
  - Python allows you to assign a single value to several variables simultaneously
  - Example:
    - a = b = c = 1
    - a,b,c = 1,2,"john"
- Code
  - **x=y=z=50** **a,b,c=5,10,15**
  - **print x** **print a**
  - **print y** **print b**
  - **print z** **print c**

# Datatypes

- Python has five standard data types
    - Numbers
    - String
    - List
    - Tuple
    - Dictionary

# Datatypes

- Python has five standard data types
  - **Numbers**: It is used to store numerical values.
  - **Ex**
    - **var1 = 1**
    - **var2 = 10**
  - **String**: It is a contiguous set of characters represented in the quotation marks
  - **Code**
    - **str = 'Hello World!'**
    - **print str # Prints complete string**
    - **print str[0] # Prints first character of the string**
    - **print str[2:5] # Prints characters starting from 3rd to 5th**
    - **print str[2:] # Prints string starting from 3rd character**
    - **print str * 2 # Prints string two times**
    - **print str + "TEST" # Prints concatenated string**

# Datatypes

- Python has five standard data types
  - **List**
    - Most versatile of Python's data types.
    - A list contains items separated by commas and enclosed within square brackets ([]).
    - To some extent, lists are similar to arrays in C, JAVA
    - One difference between them is that all the items belonging to a list can be of different data type.
  - **Code**
    - **list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]**
    - **tinylist = [123, 'john']**

    - **print list # Prints complete list**
    - **print list[0] # Prints first element of the list**
    - **print list[1:3] # Prints elements starting from 2nd till 3rd**
    - **print list[2:] # Prints elements starting from 3rd element**
    - **print tinylist * 2 # Prints list two times**
    - **print list + tinylist # Prints concatenated lists**

# Datatypes

- Python has five standard data types
  - Tuple
    - Another sequence data type that is similar to the list
    - A tuple consists of a number of values separated by commas and enclosed within parentheses.

  - Difference B/w List and Tuple.
    - Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed,
    - Tuples are enclosed in parentheses ( ( ) ) and cannot be updated.
    - Tuples can be thought of as read-only lists.

# Datatypes

- Code:
  - **tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )**
  - **tinytuple = (123, 'john')**

  - **print tuple  # Prints complete list**
  - **print tuple[0]  # Prints first element of the list**
  - **print tuple[1:3]  # Prints elements starting from 2nd till 3rd**
  - **print tuple[2:]  # Prints elements starting from 3rd element**
  - **print tinytuple * 2  # Prints list two times**
  - **print tuple + tinytuple # Prints concatenated lists**

- The following code is invalid with tuple.

  - **tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )**
  - **list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]**
  - **tuple[2] = 1000  # Invalid syntax with tuple**
  - **list[2] = 1000  # Valid syntax with list**

# Datatypes

- Python has five standard data types
  - Dictionary
    - Python's dictionaries are kind of hash table type.
    - They work like key-value pairs.
    - A dictionary key can be almost any Python type, but are usually numbers or strings. Values can be any arbitrary Python object.
    - Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).
  - Example:
    - **dict = {}**
    - **dict['one'] = "This is one"**
    - **dict[2] = "This is two"**

    - **tinydict = {'name': 'john','code':6734, 'dept': 'sales'}**
    - **print dict['one'] # Prints value for 'one' key**
    - **print dict[2] # Prints value for 2 key**
    - **print tinydict # Prints complete dictionary**
    - **print tinydict.keys() # Prints all the keys**

# Operator

Def: Operators are the constructs which can manipulate the value of operands.

Different Operators:

- **Arithmetic Operators**
  - Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.
- **Comparison (Relational) Operators**
  - Comparison operators are used to compare values. It either returns True or False
- **Assignment Operators**
  - Assignment operators are used in Python to assign values to variables.
- **Logical Operators**
  - Logical operators are the and, or, not operators.
- **Bitwise Operators**
  - Act on operands as if they were string of binary digits. It operates bit by bit, hence the name.
- **Membership Operators**
  - They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).
- **Identity Operators**
  - They are used to check if two values (or variables) are located on the same part of the memory

# Operators

- Arithmetic Operators (a=10, b=20)
  - + Addition a + b = 30
  - - Subtraction a – b = -10
  - * Multiplication a * b = 200
  - / Division b / a = 2
  - % Modulus b % a = 0
  - ** Exponent a**b =10 to the power 20
  - // Floor Division 9//2 = 4

# Operators

- Arithmetic Operators (Code)
  - **x = 15**
  - **y = 4**
  - **# Output: x + y = 19**
  - **print('x + y =',x+y)**
  - **# Output: x - y = 11**
  - **print('x - y =',x-y)**
  - **# Output: x * y = 60**
  - **print('x * y =',x*y)**
  - **# Output: x / y = 3.75**
  - **print('x / y =',x/y)**
  - **# Output: x // y = 3**
  - **print('x // y =',x//y)**
  - **# Output: x ** y = 50625**
  - **print('x ** y =',x**y)**

# Operators

- Comparison Operators (a=10, b=20)
  - == Equal to (a == b) is not true.
  - != Not Equals (a !=b) is true
  - > Greater Than (a > b) is not true.
  - < Less Than (a < b) is true.
  - >= Greater equal (a >= b) is not true.
  - <= Lesser equal (a <= b) is true.

# Operators

- Comparison Operators (a=10, b=20)

**x = 10**

- **y = 12**

- **# Output: x > y is False**
- **print('x > y  is',x>y)**

- **# Output: x < y is True**
- **print('x < y  is',x<y)**

- **# Output: x == y is False**
- **print('x == y is',x==y)**

- **# Output: x != y is True**
- **print('x != y is',x!=y)**

- **# Output: x >= y is False**
- **print('x >= y is',x>=y)**

- **# Output: x <= y is True**
- **print('x <= y is',x<=y)**

# Operators

- Assignment Operators (a=10, b=20)
  - = Assigns c = a + b assigns value of a + b into c
  - += Add AND c += a is equivalent to c = c + a
  - -= Subtract AND c -= a is equivalent to c = c - a
  - *= Multiply AND c *= a is equivalent to c = c * a
  - /= Divide AND c /= a is equivalent to c = c / a
  - %= Modulus AND c %= a is equivalent to c = c % a
  - **= Exponent AND c **= a is equivalent to c = c ** a
  - //= Floor Division c //= a is equivalent to c = c // a

# Operators

- Bitwise Operators (a=10 (0000 1010), b=4 (0000 0100))
  - **& Bitwise AND** (a & b) = 0 (0000 0000)
    - Def: The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.
  - **| Bitwise OR** (a | b) = 14 (0000 1110)
    - Def: The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1.
  - **^ Bitwise XOR** (a ^ b) = 14 (0000 1110)
    - Def: The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite.
  - **~ Bitwise NOT** ~a = -11 (1111 0101)
    - Def: Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1.
  - **<< Bitwise Left Shift** a<< 2 = 40 (0010 1000)
    - Def: Left shift operator shifts all bits towards left by certain number of specified bits.
  - **>> Bitwise Right Shift** a>> 2 = 2 (0000 0010)
    - Def: Right shift operator shifts all bits towards right by certain number of specified bits.

# Operators

- Membership Operators
  - in
    - x in y, here in results in a 1 if x is a member of sequence y.
  - not in
    - x not in y, here not in results in a 1 if x is not a member of sequence y.

# Operators

- Membership Operators

**x = 'Hello world'**

- **y = {1:'a',2:'b'}**

- **# Output: True**
- **print('H' in x)**

- **# Output: True**
- **print('hello' not in x)**

- **# Output: True**
- **print(1 in y)**

- **# Output: False**
- **print('a' in y)**

# Operators

- Identity Operators
  - is
    - x is y, here is results in 1 if id(x) equals id(y).
  - is not
    - x is not y, here is not results in 1 if id(x) is not equal to id(y).

# Operators

- Identity Operators
  - **x1 = 5**
  - **y1 = 5**
  - **x2 = 'Hello'**
  - **y2 = 'Hello'**
  - **x3 = [1,2,3]**
  - **y3 = [1,2,3]**

  - **# Output: False**
  - **print(x1 is not y1)**

  - **# Output: True**
  - **print(x2 is y2)**

  - **# Output: False**
  - **print(x3 is y3)**

# Conditional Statements

- **if statements**
  - An if statement consists of a boolean expression followed by one or more statements.
  - Syntax
    - **if expression:**
    - **  statement(s)**

# Conditional Statements

- if statements

  Code:

  - **num = 3**
  - **if num > 0:**
  - **    print(num, "is a positive number.")**
  - **print("This is always printed.")**

  - **num = -1**
  - **if num > 0:**
  - **    print(num, "is a positive number.")**
  - **print("This is also always printed.")**

# Conditional Statements

- **if...else statements**
  - An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.
  - Syntax:
    - **if expression:**
    - **  statement(s)**
    - **else:**
    - **  statement(s)**

# Conditional Statements

- If-else statements

  Code:

  - **num = 3**

  - **# Try these two variations as well.**
  - **# num = -5**
  - **# num = 0**

  - **if num >= 0:**
  - **    print("Positive or Zero")**
  - **else:**
  - **    print("Negative number")**

# Conditional Statements

- elif statements

  Code:

  - **num = 3.4**
  - 
  - **# Try these two variations as well:**
  - **# num = 0**
  - **# num = -4.5**

  - **if num > 0:**
  - **    print("Positive number")**
  - **elif num == 0:**
  - **    print("Zero")**
  - **else:**
  - **    print("Negative number")**

# Conditional Statements

- **nested if statements**
  - You can use one if or else if statement inside another if or else if statement(s).
  - Syntax:
    - **if expression1:**
    - **  statement(s)**
    - **  if expression2:**
    - **  statement(s)**
    - **  elif expression3:**
    - **  statement(s)**
    - **  else:**
    - **  statement(s)**
    - **elif expression4:**
    - **  statement(s)**
    - **else:**
    - **  statement(s)**

# Conditional Statements

- Nested if statements

  Code:

```
num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

# Looping

- for loop
  - Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
  - Syntax
    - for iterating_var in sequence:
    - statements(s)

# Looping

- for loop
  - Code:
    - for letter in 'Python':  # First Example
    - print 'Current Letter :', letter
    - print "Good bye!"

# Looping

- for loop: Iterating by Sequence Index
  - Code:
    - # Program to find the sum of all numbers stored in a list
    - 
    - # List of numbers
    - numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
    - 
    - # variable to store the sum
    - sum = 0
    - 
    - # iterate over the list
    - for val in numbers:
    - 	sum = sum+val
    - 
    - # Output: The sum is 48
    - print("The sum is", sum)

# Looping

- for loop: Using else Statement with Loops
  - Code:
    - num1=15
    - for num in range(10,20):  #to iterate between 10 to 20
    - if num<num1:
      - Print 'yes'
    - else:  # else part of the loop
    - print num, 'no'

# Looping

- Range Function: Much more simpler version of for loop, where we can generate a sequence of numbers using range() function
  - Code:
    - **# Output: range(0, 10)**
    - **print(range(10))**
    - 
    - **# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**
    - **print(list(range(10)))**
    - 
    - **# Output: [2, 3, 4, 5, 6, 7]**
    - **print(list(range(2, 8)))**
    - 
    - **# Output: [2, 5, 8, 11, 14, 17]**
    - **print(list(range(2, 20, 3)))**

# Looping

- while loop
  - Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
  - Syntax
    - while expression:
    - statement(s)

# Looping

- while loop:
  - Code:
    - count = 0
    - while (count < 9):
    - print 'The count is:', count
    - count = count + 1
    -
    - print "Good bye!"

# Looping

- while loop: Using else Statement with Loops
  - Code:
    - count = 0
    - while count < 5:
    - print count, " is  less than 5"
    - count = count + 1
    - else:
    - print count, " is not less than 5"

# Looping

- nested loop
  - You can use one or more loop inside any another while, for or do..while loop.
  - Syntax
    - while expression:
    - while expression:
    - statement(s)
    - statement(s)

# Looping

- nested loop:
  - Code:
    - i = 2
    - while(i < 100):
    -  j = 2
    -  while(j <= 50)):
    -  j = j + 1
    -  print j
    -  i = i + 1
    -  print i

    - print "Good bye!"

# Control Statements

- **Break**
  - Terminates the loop statement and transfers execution to the statement immediately following the loop.
  - The most common use for break is when some external condition is triggered requiring a exit from a loop.
- Code:
  - **for letter in 'Python':  # First Example**
  - **  if letter == 'h':**
  - **  break**
  - **  print 'Current Letter :', letter**
  - 
  - **var = 10  # Second Example**
  - **while var > 0:**
  - **  print 'Current variable value :', var**
  - **  var = var -1**
  - **  if var == 5:**
  - **  break**
  - 
  - **print "Good bye!"**

# Control Statements

- **Continue**
  - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
  - The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- Code:
  - **for letter in 'Python':  # First Example**
  - **  if letter == 'h':**
  - **  continue**
  - **  print 'Current Letter :', letter**
  - 
  - **var = 10  # Second Example**
  - **while var > 0:**
  - **  var = var -1**
  - **  if var == 5:**
  - **  continue**
  - **  print 'Current variable value :', var**
  - **print "Good bye!"**

# Control Statements

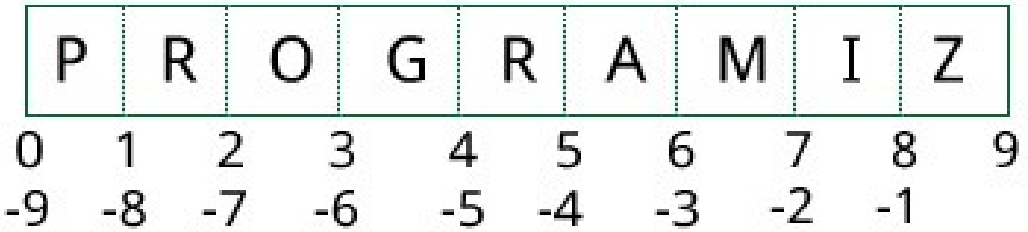
- **Pass**
  - The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute..
  - pass is just a placeholder for functionality to be added later.
- Code:
  - **sequence = {'p', 'a', 's', 's'}**
  - **for val in sequence:**
  - **    pass**

# String Manipulation

- How to create a string?
  - Strings can be created by enclosing characters inside a single or double and even triple quotes.
    - **# 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)**

# String Manipulation

- How to access characters in a string?
  - We can access individual characters using indexing and a range of characters using slicing. Index starts from 0.
    - **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])**



# String Manipulation

- How to access characters in a string?
  - Two types of errors when accessing characters in a string
    - **IndexError**: Trying to access a character out of index srange
    - **TypeError**: Using float or other types to access the data
      - **# index must be in range**
      - **>>> my_string[15]**
      - **...**
      - **IndexError: string index out of range**
      -
      - **# index must be an integer**
      - **>>> my_string[1.5]**
      - **...**
      - **TypeError: string indices must be integers**

# String Manipulation

- How to change or delete a string?
  - Strings are immutable. This means that elements of a string cannot be changed once it has been assigned. We can simply reassign different strings to the same name.
    - **>>> my_string = 'programiz'**
    - **>>> my_string[5] = 'a'**
    - **TypeError: 'str' object does not support item assignment**
    - **>>> my_string = 'Python'**
    - **>>> my_string**
    - **'Python'**

# String Manipulation

- How to change or delete a string?
  - We cannot delete or remove characters from a string. But deleting the string entirely is possible using the keyword del
    - **>>> del my_string[1]**
    - **TypeError: 'str' object doesn't support item deletion**
    - **>>> del my_string**
    - **>>> print my_string**
    - **NameError: name 'my_string' is not defined**

# String Manipulation

- How to change or delete a string?
  - You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.
    - **var1 = 'Hello World!'**
    - **var2=var1[:6] + 'Python'**
    - **print "Updated String :- ", var2**

# String Manipulation

- Python String Operations
  - Concatenation of Two or More Strings
    - Joining of two or more strings into a single one is called concatenation.
    - The + operator does this in Python.

    - **str1 = 'Hello'**
    - **str2 ='World!'**
    -
    - **# using +**
    - **print('str1 + str2 = ', str1 + str2)**
    -
    - **# using ***
    - **print('str1 * 3 =', str1 * 3)**

# String Manipulation

- Python String Operations
  - Concatenation of Two or More Strings
    - Writing two string literals together also concatenates them like + operator.
    - If we want to concatenate strings in different lines, we can use parentheses.

    - **>>> # two string literals together**
    - **>>> 'Hello ''World!'**
    - **'Hello World!'**

    - **>>> # using parentheses**
    - **>>> s = ('Hello '**
    - **...  'World')**
    - **>>> s**
    - **'Hello World'**

# String Manipulation

- Python String Operations
  - Iterating Through String
    - Using for loop we can iterate through a string. Here is an example to count the number of 'l' in a string.

```
count = 0
for letter in 'Hello World':
    if(letter == 'l'):
        count += 1
print(count,'letters found')
```

# String Manipulation

- Python String Operations
  - String Membership Test
    - We can test if a sub string exists within a string or not, using the keyword in.

    - **>>> 'a' in 'program'**
    - **True**
    - **>>> 'at' not in 'battle'**
    - **False**

# String Manipulation

- Python Built-in functions
  - commonly used ones are enumerate() and len()
    - The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs.
    - len() returns the length (number of characters) of the string.

    - **str = 'cold'**
    - **# enumerate()**
    - **list_enumerate = list(enumerate(str))**
    - **print('list(enumerate(str) = ', list_enumerate)**

    - **#character count**
    - **print('len(str) = ', len(str))**
    - **False**

# String Manipulation

- Python String Formatting
  - Escape Sequence
    - An escape sequence starts with a backslash and is interpreted differently. If we use single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes.

    - **# using triple quotes**
    - **print('''He said, "What's there?"''')**
    - 
    - **# escaping single quotes**
    - **print('He said, "What\'s there?"')**
    - 
    - **# escaping double quotes**
    - **print("He said, \"What's there?\"")**

# String Manipulation

- Python String Formatting
  - String format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family.

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

# String Manipulation

- Python String Methods
  - **Refer the following link**
  - **https://docs.python.org/2/library/stdtypes.html#string-methods**

# Lists

Def:

- A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements.
- Each element or value that is inside of a list is called an item.
- Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ].

# Lists

## Creating a list:

- Creating a list is as simple as putting different comma-separated values between square brackets.
- It can have any number of items and they may be of different types (integer, float, string etc.).
- Example:
  - list1 = ['physics', 'chemistry', 1997, 2000];
  - list2 = [1, 2, 3, 4, 5 ];
  - list3 = ["a", "b", "c", "d"]
- Also, a list can even have another list as an item. This is called nested list.
  - # nested list
  - my_list = ["mouse", [8, 4, 6], ['a']]

# Lists

How to access elements from a list?

- List Index: We can use the index operator [] to access an item in a list. Index starts from 0.
- Trying to access an element other that this will raise an IndexError.
- The index must be an integer.
- We can't use float or other types, this will result into TypeError.

# Lists

## How to access elements from a list?

- Code
- my_list = ['p','r','o','b','e']
- # Output: p
- print(my_list[0])

- # Output: o
- print(my_list[2])

- # Output: e
- print(my_list[4])

- # Error! Only integer can be used for indexing
- # my_list[4.0]

- # Nested List
- n_list = ["Happy", [2,0,1,5]]

- # Nested indexing

- # Output: a
- print(n_list[0][1])

- # Output: 5
- print(n_list[1][3])

# Lists

How to access elements from a list?

- **Negative indexing**: Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.
  - my_list = ['p','r','o','b','e']
  - # Output: e
  - print(my_list[-1])
  - # Output: p
  - print(my_list[-5])

# Lists

- How to slice lists in Python?
  - We can access a range of items in a list by using the slicing operator (colon).
    - my_list = ['p','r','o','g','r','a','m','i','z']
    - # 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[:])

# Lists

- How to change or add elements to a list?
  - List are mutable, meaning, their elements can be changed unlike string or tuple.
  - We can use assignment operator (=) to change an item or a range of items.
    - # mistake values
    - odd = [2, 4, 6, 8]
    - # change the 1st item
    - odd[0] = 1
    - # Output: [1, 4, 6, 8]
    - print(odd)
    - # change 2nd to 4th items
    - odd[1:4] = [3, 5, 7]
    - # Output: [1, 3, 5, 7]
    - print(odd)

# Lists

- How to change or add elements to a list?
  - We can add one item to a list using append() method or add several items using extend() method.
    - odd = [1, 3, 5]
    - odd.append(7)
    - # Output: [1, 3, 5, 7]
    - print(odd)
    - odd.extend([9, 11, 13])
    - # Output: [1, 3, 5, 7, 9, 11, 13]
    - print(odd)

# Lists

- How to change or add elements to a list?
  - We can also use + operator to combine two lists. This is also called concatenation.
  - The * operator repeats a list for the given number of times.
    - odd = [1, 3, 5]
    -
    - # Output: [1, 3, 5, 9, 7, 5]
    - print(odd + [9, 7, 5])
    -
    - #Output: ["re", "re", "re"]
    - print(["re"] * 3)

# Lists

- How to change or add elements to a list?
  - We can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.
    - odd = [1, 9]
    - odd.insert(1,3)
    - # Output: [1, 3, 9]
    - print(odd)
    - odd[2:2] = [5, 7]
    - # Output: [1, 3, 5, 7, 9]
    - print(odd)

# Lists

- How to delete or remove elements from a list?
  - We can delete one or more items from a list using the keyword del. It can even delete the list entirely.
    - my_list = ['p','r','o','b','l','e','m']
    - # delete one item
    - del my_list[2]
    - # Output: ['p', 'r', 'b', 'l', 'e', 'm']
    - print(my_list)
    - # delete multiple items
    - del my_list[1:5]
    - # Output: ['p', 'm']
    - print(my_list)
    - # delete entire list
    - del my_list
    - # Error: List not defined
    - print(my_list)

# Lists

- How to delete or remove elements from a list?
  - We can use remove() method to remove the given item or pop() method to remove an item at the given index.
  - The pop() method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure).
  - We can also use the clear() method to empty a list.

# Lists

- How to delete or remove elements from a list?
  - my_list = ['p','r','o','b','l','e','m']
  - my_list.remove('p')
  - # Output: ['r', 'o', 'b', 'l', 'e', 'm']
  - print(my_list)
  - # Output: 'o'
  - print(my_list.pop(1))
  - # Output: ['r', 'b', 'l', 'e', 'm']
  - print(my_list)
  - # Output: 'm'
  - print(my_list.pop())
  - # Output: ['r', 'b', 'l', 'e']
  - print(my_list)
  - my_list.clear()
  - # Output: []
  - print(my_list)

# Lists

- How to delete or remove elements from a list?
  - we can also delete items in a list by assigning an empty list to a slice of elements.
  - **>>> my_list = ['p','r','o','b','l','e','m']**
  - **>>> my_list[2:3] = []**
  - **>>> my_list**
  - **['p', 'r', 'b', 'l', 'e', 'm']**
  - **>>> my_list[2:5] = []**
  - **>>> my_list**
  - **['p', 'r', 'm']**

# Lists

- Python List Methods
  - They are accessed as list.method().
  - **Python List Methods**
  - **append() - Add an element to the end of the list**
  - **extend() - Add all elements of a list to the another list**
  - **insert() - Insert an item at the defined index**
  - **remove() - Removes an item from the list**
  - **pop() - Removes and returns an element at the given index**
  - **clear() - Removes all items from the list**
  - **index() - Returns the index of the first matched item**
  - **count() - Returns the count of number of items passed as an argument**
  - **sort() - Sort items in a list in ascending order**
  - **reverse() - Reverse the order of items in the list**
  - **copy() - Returns a shallow copy of the list**

# Lists

- Python List Methods
  - my_list = [3, 8, 1, 6, 0, 8, 4]
  -
  - # Output: 1
  - print(my_list.index(8))
  -
  - # Output: 2
  - print(my_list.count(8))
  -
  - my_list.sort()
  -
  - # Output: [0, 1, 3, 4, 6, 8, 8]
  - print(my_list)
  -
  - my_list.reverse()
  -
  - # Output: [8, 8, 6, 4, 3, 1, 0]
  - print(my_list)

# Lists

- Built-in Functions with List
  - Built-in functions like all(), any(), enumerate(), len(), max(), min(), list(), sorted() etc. are commonly used with list to perform different tasks.
  - **all()  Return True if all elements of the list are true (or if the list is empty).**
  - **any() Return True if any element of the list is true. If the list is empty, return False.**
  - **enumerate()  Return an enumerate object. It contains the index and value of all the items of list as a tuple.**
  - **len()  Return the length (the number of items) in the list.**
  - **list()  Convert an iterable (tuple, string, set, dictionary) to a list.**
  - **max()  Return the largest item in the list.**
  - **min() Return the smallest item in the list**
  - **sorted() Return a new sorted list (does not sort the list itself).**
  - **sum()  Return the sum of all elements in the list.**

# Tuple

- A tuple is a sequence of immutable Python objects.
- The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas in a list, elements can be changed.
- Advantages of Tuple over List
  - Since, tuples are quite similiar to lists, both of them are used in similar situations as well.
  - However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:
  - We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
  - Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
  - If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

# Tuple

- Creating a Tuple
  - A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma.
  - A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

# Tuple

- Creating a Tuple
  - # empty tuple
  - # Output: ()
  - my_tuple = ()
  - print(my_tuple)
  -
  - # tuple having integers
  - # Output: (1, 2, 3)
  - my_tuple = (1, 2, 3)
  - print(my_tuple)
  -
  - # tuple with mixed datatypes
  - # Output: (1, "Hello", 3.4)
  - my_tuple = (1, "Hello", 3.4)
  - print(my_tuple)
  -
  - # nested tuple
  - # Output: ("mouse", [8, 4, 6], (1, 2, 3))
  - my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
  - print(my_tuple)
  -
  - # tuple can be created without parentheses
  - # also called tuple packing
  - # Output: 3, 4.6, "dog"
  -
  - my_tuple = 3, 4.6, "dog"
  - print(my_tuple)

# Tuple

- Accessing Elements in a Tuple
  - Indexing
    - my_tuple = ('p','e','r','m','i','t')
    - # Output: 'p'
    - print(my_tuple[0])
    - # Output: 't'
    - print(my_tuple[5])
    - # nested tuple
    - n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
    - # nested index
    - # Output: 's'
    - print(n_tuple[0][3])
    - # nested index
    - # Output: 4
    - print(n_tuple[1][1])

# Tuple

- Accessing Elements in a Tuple
  - Negative Indexing
    - my_tuple = ('p','e','r','m','i','t')
    -
    - # Output: 't'
    - print(my_tuple[-1])
    -
    - # Output: 'p'
    - print(my_tuple[-6])

# Tuple

- Accessing Elements in a Tuple
  - Slicing
    - my_tuple = ('p','r','o','g','r','a','m','i','z')
    -
    - # elements 2nd to 4th
    - # Output: ('r', 'o', 'g')
    - print(my_tuple[1:4])
    -
    - # elements beginning to 2nd
    - # Output: ('p', 'r')
    - print(my_tuple[:-7])
    -
    - # elements 8th to end
    - # Output: ('i', 'z')
    - print(my_tuple[7:])
    -
    - # elements beginning to end
    - # Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
    - print(my_tuple[:])

# Tuple

- Changing a Tuple
  - Unlike lists, tuples are immutable.
  - This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.
    - **my_tuple = (4, 2, 3, [6, 5])**
    - 
    - **# we cannot change an element**
    - **# If you uncomment line 8**
    - **# you will get an error:**
    - **# TypeError: 'tuple' object does not support item assignment**
    - 
    - **#my_tuple[1] = 9**
    - 
    - **# but item of mutable element can be changed**
    - **# Output: (4, 2, 3, [9, 5])**
    - **my_tuple[3][0] = 9**
    - **print(my_tuple)**
    - 
    - **# tuples can be reassigned**
    - **# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')**
    - **my_tuple = ('p','r','o','g','r','a','m','i','z')**
    - **print(my_tuple)**

# Tuple

- Changing a Tuple
  - We can use + operator to combine two tuples. This is also called concatenation.
  - We can also repeat the elements in a tuple for a given number of times using the * operator.
    - # Concatenation
    - # Output: (1, 2, 3, 4, 5, 6)
    - print((1, 2, 3) + (4, 5, 6))
    -
    - # Repeat
    - # Output: ('Repeat', 'Repeat', 'Repeat')
    - print(("Repeat",) * 3)

# Tuple

- Deleting a Tuple
  - We cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.
  - But deleting a tuple entirely is possible using the keyword del.
    - my_tuple = ('p','r','o','g','r','a','m','i','z')
    - 
    - # can't delete items
    - # if you uncomment line 8,
    - # you will get an error:
    - # TypeError: 'tuple' object doesn't support item deletion
    - 
    - #del my_tuple[3]
    - 
    - # can delete entire tuple
    - # NameError: name 'my_tuple' is not defined
    - del my_tuple
    - my_tuple

# Tuple

- Python Tuple Methods
  - Methods that add items or remove items are not available with tuple. Only the following two methods are available.
  - count(x) Return the number of items that is equal to x
  - index(x) Return index of first item that is equal to x
    - my_tuple = ('a','p','p','l','e',)
    -
    - # Count
    - # Output: 2
    - print(my_tuple.count('p'))
    -
    - # Index
    - # Output: 3
    - print(my_tuple.index('l'))

# Tuple

- Other Tuple Operations
  - Tuple Membership Test
    - my_tuple = ('a','p','p','l','e',)
    -
    - # In operation
    - # Output: True
    - print('a' in my_tuple)
    -
    - # Output: False
    - print('b' in my_tuple)
    -
    - # Not in operation
    - # Output: True
    - print('g' not in my_tuple)

# Tuple

- Other Tuple Operations
  - Iterating Through a Tuple
    - # Output:
    - # Hello John
    - # Hello Kate
    - for name in ('John','Kate'):
    - print("Hello",name)

# Tuple

- Built-in Functions with Tuple
  - all()  Return True if all elements of the tuple are true (or if the tuple is empty).
  - any() Return True if any element of the tuple is true. If the tuple is empty, return False.
  - enumerate()  Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
  - len()  Return the length (the number of items) in the tuple.
  - max()  Return the largest item in the tuple.
  - min() Return the smallest item in the tuple
  - sorted() Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
  - sum()  Retrun the sum of all elements in the tuple.
  - tuple()  Convert an iterable (list, string, set, dictionary) to a tuple.

# Dictionary

- Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

- Dictionaries are optimized to retrieve values when the key is known.

# Dictionary

- How to create a dictionary?
  - Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.
  - An item has a key and the corresponding value expressed as a pair, key: value.
  - While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

# Dictionary

- How to create a 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

- How to access elements from a dictionary?
  - While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the get() method.
  - The difference while using get() is that it returns None instead of KeyError, if the key is not found.
    - **my_dict = {'name':'Jack', 'age': 26}**
    - 
    - **# Output: Jack**
    - **print(my_dict['name'])**
    - 
    - **# Output: 26**
    - **print(my_dict.get('age'))**
    - 
    - **# Trying to access keys which doesn't exist throws error**
    - **# my_dict.get('address')**
    - **# my_dict['address']**

# Dictionary

- How to change or add elements in a dictionary?
  - Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.
  - If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.
    - **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'**
    - 
    - **# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}**
    - **print(my_dict)**

# Dictionary

- How to delete or remove elements from a dictionary?
  - We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.
  - The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the clear() method.
  - We can also use the del keyword to remove individual items or the entire dictionary itself.

# Dictionary

- How to delete or remove elements from a dictionary?
  - # create a dictionary
  - squares = {1:1, 2:4, 3:9, 4:16, 5:25}
  -
  - # remove a particular item
  - # Output: 16
  - print(squares.pop(4))
  -
  - # Output: {1: 1, 2: 4, 3: 9, 5: 25}
  - print(squares)
  -
  - # remove an arbitrary item
  - # Output: (1, 1)
  - print(squares.popitem())
  -
  - # Output: {2: 4, 3: 9, 5: 25}
  - print(squares)
  -
  - # delete a particular item
  - del squares[5]
  -
  - # Output: {2: 4, 3: 9}
  - print(squares)
  -
  - # remove all items
  - squares.clear()
  -
  - # Output: {}
  - print(squares)
  -
  - # delete the dictionary itself
  - del squares
  -
  - # Throws Error
  - # print(squares)

# Dictionary

- Python Dictionary Methods

  Method  Description
- clear()  Remove all items form the dictionary.
- copy()  Return a shallow copy of the dictionary.
- fromkeys(seq[, v])  Return a new dictionary with keys from seq and value equal to v (defaults to None).
- get(key[,d])  Return the value of key. If key doesnot exit, return d (defaults to None).
- items() Return a new view of the dictionary's items (key, value).
- keys()  Return a new view of the dictionary's keys.
- pop(key[,d]) Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
- popitem() Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
- setdefault(key[,d])  If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
- update([other]) Update the dictionary with the key/value pairs from other, overwriting existing keys.
- values()  Return a new view of the dictionary's values

# Dictionary

- Other Dictionary Operations
  - Dictionary Membership Test

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
•
• # Output: True
• print(1 in squares)
•
• # Output: True
• print(2 not in squares)
•
• # membership tests for key only not value
• # Output: False
• print(49 in squares)
```

# Dictionary

- Other Dictionary Operations
  - Iterating Through a Dictionary
    - squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
    - for i in squares:
    - print(squares[i])

# Dictionary

- Built-in Functions with Dictionary
  - Function Description
  - all() Return True if all keys of the dictionary are true (or if the dictionary is empty).
  - any() Return True if any key of the dictionary is true. If the dictionary is empty, return False.
  - len() Return the length (the number of items) in the dictionary.
  - cmp() Compares items of two dictionaries.
  - sorted() Return a new sorted list of keys in the dictionary.

# Functions

- Def: In Python, function is a group of related statements that perform a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes code reusable.

# Functions

- **Syntax**:
  - **def function_name(parameters):**
    - **"""docstring"""**
    - **statement(s)**
    - **return (s)**
- Keyword **def** marks the start of function header.
- A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does.
- An optional return statement to return a value from the function.

# Functions

- Example of a function:
  - def greet(name):
    - """This function greets to
    - the person passed in as
    - parameter"""
    - print("Hello, " + name + ". Good morning!")

- **Function Call**
  - Once we have defined a function, we can call it from another function, program or even the Python prompt
    - >>> greet('Paul')
    - Hello, Paul. Good morning!

# Functions

- The return statement
  - The return statement is used to exit a function and go back to the place from where it was called.
  - Syntax of return
    - return [expression_list]
  - This statement can contain expression which gets evaluated and the value is returned.
  - If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.
  - For example:
    - >>> **print(greet("May"))**
    - Hello, May. Good morning!
    - None

# Functions

- The return statement
  - Example of return
    - **def absolute_value(num):**
      - **"""This function returns the absolute**
      - **value of the entered number"""**

      - **if num >= 0:**
      - **return num**
      - **else:**
      - **return -num**

    - **# Output: 2**
    - **print(absolute_value(2))**

    - **# Output: 4**
    - **print(absolute_value(-4))**

# Functions

- How Function works in Python?
  - Scope and Lifetime of variables
    - Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.
    - Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
    - They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

# Functions

- How Function works in Python?
  - Example:
    - **def my_func():**
      - **x = 10**
      - **print("Value inside function:",x)**
    - **x = 20**
    - **my_func()**
    - **print("Value outside function:",x)**

- On the other hand, variables outside of the function are visible from inside. They have a global scope.
- We can read these values from inside the function but cannot change (write) them.
- In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

# Functions

- Types of Functions
  - Built-in functions - Functions that are built into Python.
  - User-defined functions - Functions defined by the users themselves.

# Functions

- Function Arguments
  - In Python, you can define a function that takes variable number of arguments.
  - Example:
    - **def greet(name,msg):**
    - **"""This function greets to**
    - **the person with the provided message"""**
    - **print("Hello",name + ', ' + msg)**
    -
    - **greet("Monica","Good morning!")**

# Functions

- Function Arguments
  - **Here, the function greet() has two parameters.**
  - Since, we have called this function with two arguments, it runs smoothly and we do not get any error.
  - If we call it with different number of arguments, the interpreter will complain.
  - >>> greet("Monica")  # only one argument
  - TypeError: greet() missing 1 required positional argument: 'msg'
  - >>> greet()  # no arguments
  - TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'

# Functions

- Variable Function Arguments
  - In Python there are other ways to define a function which can take variable number of arguments.
  - Python Default Arguments
    - Function arguments can have default values in Python.
    - We can provide a default value to an argument by using the assignment operator (=).
  - Python Keyword Arguments
    - When we call a function with some values, these values get assigned to the arguments according to their position.
  - Python Arbitrary Arguments
    - Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

# Functions

- Variable Function Arguments
  - Python Default Arguments
    - def greet(name, msg = "Good morning!"):
    - print("Hello",name + ', ' + msg)
    -
    - greet("Kate")
    - greet("Bruce","How do you do?")
  - In this function, the parameter name does not have a default value and is required (mandatory) during a call.
  - On the other hand, the parameter msg has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

# Functions

- Variable Function Arguments
  - Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.
  - def greet(msg = "Good morning!", name):
  -
  - We would get an error as:
  - SyntaxError: non-default argument follows default argument

# Functions

- Python Keyword Arguments
  - In the above function greet(), when we called it as greet("Bruce","How do you do?"), the value "Bruce" gets assigned to the argument name and similarly "How do you do?" to msg.
  - Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.
  - >>> # 2 keyword arguments
  - >>> greet(name = "Bruce",msg = "How do you do?")
  -
  - >>> # 2 keyword arguments (out of order)
  - >>> greet(msg = "How do you do?",name = "Bruce")
  -
  - >>> # 1 positional, 1 keyword argument
  - >>> greet("Bruce",msg = "How do you do?")

# Functions

- Python Keyword Arguments
  - As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.
  - Having a positional argument after keyword arguments will result into errors. For example the function call as follows:
  - greet(name="Bruce","How do you do?")
  - Will result into error as:
  - SyntaxError: non-keyword arg after keyword arg

# Functions

- Python Arbitrary Arguments
  - In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.
  - Example:
    - def greet(*names):
    - """This function greets all
    - the person in the names tuple."""

    - # names is a tuple with arguments
    - for name in names:
    - print("Hello",name)

    - greet("Monica","Luke","Steve","John")

# Functions

- Recursive Function
  - Recursion is the process of defining something in terms of itself.
  - Example:
    - # An example of a recursive function to find the factorial of a number
    - def calc_factorial(x):
    - """This is a recursive function to find the factorial of an integer"""
    - if x == 1:
    - return 1
    - else:
    - return (x * calc_factorial(x-1))

    - num = 4
    - print("The factorial of", num, "is", calc_factorial(num))

# Functions

- Recursive Function
  - In the above example, calc_factorial() is a recursive functions as it calls itself.
  - When we call this function with a positive integer, it will recursively call itself by decreasing the number.
  - Each function call multiples the number with the factorial of number 1 until the number is equal to one.
    - calc_factorial(4) # 1st call with 4
    - 4 * calc_factorial(3) # 2nd call with 3
    - 4 * 3 * calc_factorial(2) # 3rd call with 2
    - 4 * 3 * 2 * calc_factorial(1) # 4th call with 1
    - 4 * 3 * 2 * 1 # return from 4th call as number=1
    - 4 * 3 * 2 # return from 3rd call
    - 4 * 6 # return from 2nd call
    - 24 # return from 1st call

# Functions

- Recursive Function
  - **Advantages of recursion**
    - Recursive functions make the code look clean and elegant.
    - A complex task can be broken down into simpler sub-problems using recursion.
    - Sequence generation is easier with recursion than using some nested iteration.
  - **Disadvantages of recursion**
    - Sometimes the logic behind recursion is hard to follow through.
    - Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
    - Recursive functions are hard to debug.

# Functions

- **Global vs. Local variables**
  - Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
  - This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.
  - **Example**
    - **total = 0; # This is global variable.**
    - **# Function definition is here**
    - **def sum( arg1, arg2 ):**
    - **# Add both the parameters and return them."**
    - **total = arg1 + arg2; # Here total is local variable.**
    - **print "Inside the function local total : ", total**
    - **return total;**

    - **# Now you can call sum function**
    - **sum( 10, 20 );**
    - **print "Outside the function global total : ", total**

# Modules

- Def: Modules refer to a file containing Python statements and definitions.
- A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.
- We use modules to break down large programs into small manageable and organized files.
- Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

# Modules

- Creating a Module
  - Let us create a module. Type the following and save it as example.py.
    - # Python Module example
    - def add(a, b):
    - """This program adds two
    - numbers and return the result"""
    - result = a + b
    - return result
  - Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

# Modules

- Import modules in Python?
  - We can import the definitions inside a module to another module
  - We use the import keyword to do this.
  - To import our previously defined module to another module, we use
    - **>>> import example**
  - Using the module name we can access the function using dot (.) operation.
  - **For example:**
    - **>>> example.add(4,5.5)**
    - **9.5**

# Modules

- Import modules in Python?
  - There are various ways to import modules. They are listed as follows.
  - **Python import statement**
  - We can import a module using import statement and access the definitions inside it using the dot operator as described above.
  - Example:
    - **# import statement example**
    - **# to import standard module math**
    - 
    - **import math**
    - **print("The value of pi is", math.pi)**

# Modules

- Import modules in Python?
  - **Import with renaming**
  - Example:
    - **# import module by renaming it**
    - 
    - **import math as m**
    - **print("The value of pi is", m.pi)**
  - We have renamed the math module as m. This can save us typing time in some cases.
  - Note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

# Modules

- Import modules in Python?
  - **Python from...import statement**
    - We can import specific names form a module without importing the module as a whole. Here is an example.
    - Example:
      - **# import only pi from math module**
      - **from math import pi**
      - **print("The value of pi is", pi)**
    - We imported only the attribute pi form the module.
    - In such case we don't use the dot operator. We could have imported multiple attributes as follows.
      - **>>> from math import pi, e**
      - **>>> pi**
      - **3.141592653589793**
      - **>>> e**
      - **2.718281828459045**

# Modules

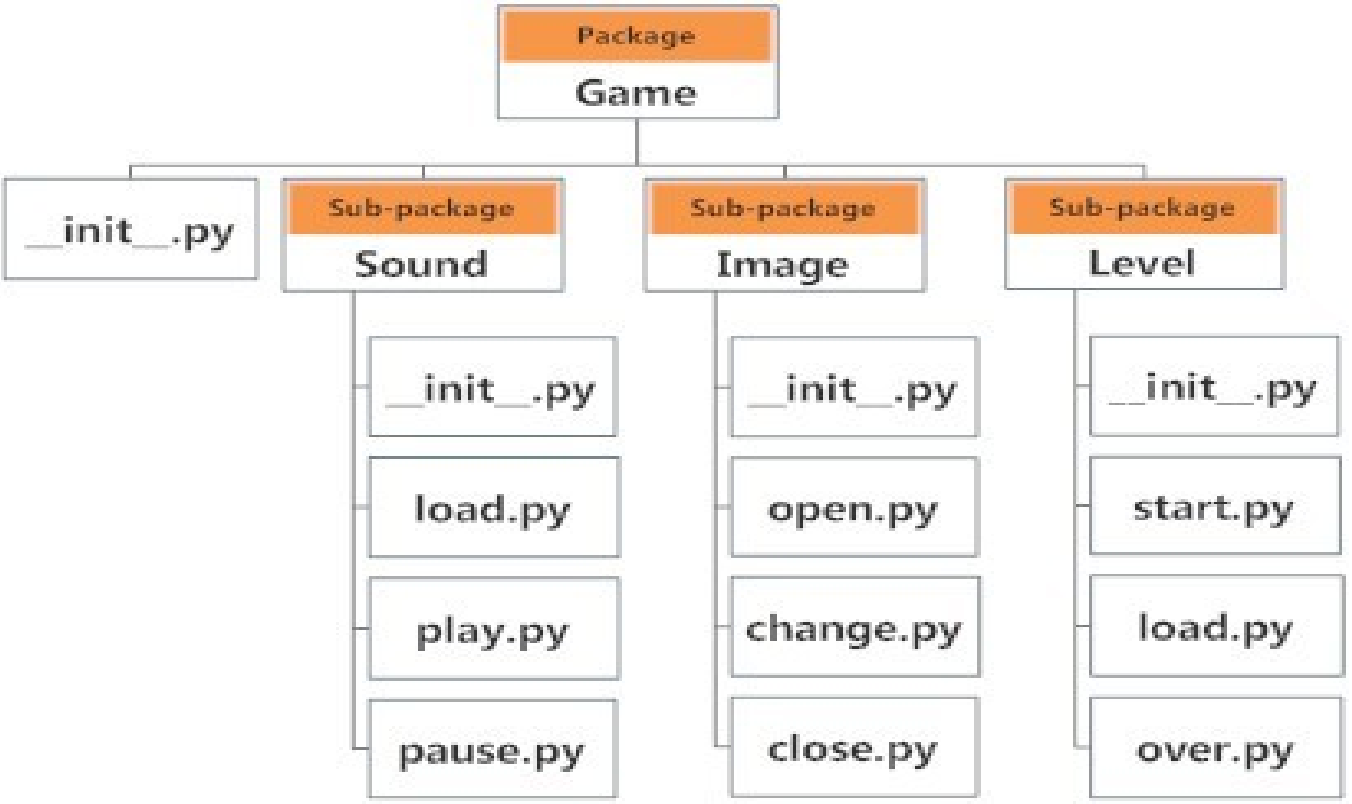
- Import modules in Python?
  - **Import all names**
    - We can import all names(definitions) form a module using the following construct.
    - Example:
      - **# import all names form**
      - **# the standard module math**
      -
      - **from math import ***
      - **print("The value of pi is", pi)**

# Packages

- We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.
- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.
- Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

# Packages

- Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.
-



# Packages

- Importing module from a package
- We can import modules from packages using the dot (.) operator.
- For example, if want to import the start module in the above example, it is done as follows.
  - **import Game.Level.start**
- Now if this module contains a function named select_difficulty(), we must use the full name to reference it.
  - **Game.Level.start.select_difficulty(2)**
- If this construct seems lengthy, we can import the module without the package prefix as follows.
  - **from Game.Level import start**
- We can now call the function simply as follows.
  - **start.select_difficulty(2)**

# Files

- File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.
- When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.
- Hence, in Python, a file operation takes place in the following order.
  - Open a file
  - Read or write (perform operation)
  - Close the file

# Files

- **Opening a file**
- Python has a built-in function open() to open a file. This function returns a file object.
  - **>>> f = open("test.txt")  # open file in current directory**
  - **>>> f = open("C:/Python33/README.txt")  # specifying full path**
- We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.
- The default is reading in text mode. In this mode, we get strings when reading from the file.
- On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

# Files

- **Python File Modes**
- 'r'  Open a file for reading. (default)
- 'w'Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
- 'x' Open a file for exclusive creation. If the file already exists, the operation fails.
- 'a' Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
- 't'  Open in text mode. (default)
- 'b' Open in binary mode.
- '+' Open a file for updating (reading and writing)

- f = open("test.txt")      # equivalent to 'r' or 'rt'
- f = open("test.txt",'w')  # write in text mode
- f = open("img.bmp",'r+b') # read and write in binary mode

# Files

- **Closing a File**
- When we are done with operations to the file, we need to properly close it.
- Closing a file will free up the resources that were tied with the file and is done using the close() method.
- Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.
  - **f = open("test.txt",encoding = 'utf-8')**
  - **# perform file operations**
  - **f.close()**

# Files

- **Writing to a File**
- In order to write into a file we need to open it in write 'w', append 'a' or exclusive creation 'x' mode.
- We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.
- Writing a string or sequence of bytes (for binary files) is done using write() method. This method returns the number of characters written to the file.
  - **with open("test.txt",'w',encoding = 'utf-8') as f:**
  - **f.write("my first file\n")**
  - **f.write("This file\n\n")**
  - **f.write("contains three lines\n")**
- This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten.

# Files

- **Reading From a File**
- To read the content of a file, we must open the file in reading mode.
- There are various methods available for this purpose. We can use the read(size) method to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file.
  - >>> f = open("test.txt",'r',encoding = 'utf-8')
  - >>> f.read(4)    # read the first 4 data
  - 'This'
  - >>> f.read(4)    # read the next 4 data
  - ' is '
  - >>> f.read()     # read in the rest till end of file
  - 'my first file\nThis file\ncontains three lines\n'
  - >>> f.read()  # further reading returns empty sting
  - ''

# Files

- **Python File Methods**
- close() Close an open file. It has no effect if the file is already closed.
- detach()  Separate the underlying binary buffer from the TextIOBase and return it.
- fileno() Return an integer number (file descriptor) of the file.
- flush()  Flush the write buffer of the file stream.
- isatty() Return True if the file stream is interactive.
- read(n) Read atmost n characters form the file. Reads till end of file if it is negative or None.
- readable()  Returns True if the file stream can be read from.
- readline(n=-1) Read and return one line from the file. Reads in at most n bytes if specified.
- readlines(n=-1)  Read and return a list of lines from the file. Reads in at most n bytes/characters if specified.
- seek(offset,from=SEEK_SET)Change the file position to offset bytes, in reference to from (start, current, end).
- seekable()  Returns True if the file stream supports random access.
- tell() Returns the current file location.
- truncate(size=None)  Resize the file stream to size bytes. If size is not specified, resize to current location.
- writable() Returns True if the file stream can be written to.
- write(s)Write string s to the file and return the number of characters written.
- writelines(lines)  Write a list of lines to the file.

# Exception Handling

- Exception
  - Python (interpreter) raises exceptions when it encounter errors.
  - Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.
    - >>> if a < 3
    - File "<interactive input>", line 1
    - if a < 3
    - ^
    - SyntaxError: invalid syntax
  - Errors can also occur at runtime and these are called exceptions.
  - for example, when a file we try to open does not exist (FileNotFoundError),
  - dividing a number by zero (ZeroDivisionError),
  - module we try to import is not found (ImportError) etc.

# Exception Handling

- Exception
  - Whenever these type of runtime error occur, Python creates an exception object.
  - If not handled properly, it prints a traceback to that error along with some details about why that error occurred.
    - **>>> 1 / 0**
    - **Traceback (most recent call last):**
    - **File "<string>", line 301, in runcode**
    - **File "<interactive input>", line 1, in <module>**
    - **ZeroDivisionError: division by zero**
    - 
    - **>>> open("imaginary.txt")**
    - **Traceback (most recent call last):**
    - **File "<string>", line 301, in runcode**
    - **File "<interactive input>", line 1, in <module>**
    - **FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'**

# Exception Handling

- Python Built-in Exceptions
  - **Exception Cause of Error**
  - **AssertionError Raised when assert statement fails.**
  - **AttributeError Raised when attribute assignment or reference fails.**
  - **EOFError Raised when the input() functions hits end-of-file condition.**
  - **FloatingPointError Raised when a floating point operation fails.**
  - **GeneratorExit Raise when a generator's close() method is called.**
  - **ImportError Raised when the imported module is not found.**
  - **IndexError Raised when index of a sequence is out of range.**
  - **KeyError Raised when a key is not found in a dictionary.**
  - **KeyboardInterrupt Raised when the user hits interrupt key (Ctrl+c or delete).**
  - **MemoryError Raised when an operation runs out of memory.**
  - **NameError Raised when a variable is not found in local or global scope.**
  - **NotImplementedError Raised by abstract methods.**
  - **OSError Raised when system operation causes system related error.**
  - **OverflowError Raised when result of an arithmetic operation is too large to be represented.**
  - **ReferenceError Raised when a weak reference proxy is used to access a garbage collected referent.**
  - **RuntimeError Raised when an error does not fall under any other category.**
  - **StopIteration Raised by next() function to indicate that there is no further item to be returned by iterator.**

# Exception Handling

- Python Custom Exceptions
  - Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.
  - However, sometimes you may need to create custom exceptions that serves your purpose.
  - In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived form this class.

# Exception Handling

- Python Custom Exceptions
  - **# define Python user-defined exceptions**
  - **class Error(Exception):**
  - **"""Base class for other exceptions"""**
  - **pass**
  - 
  - **class ValueTooSmallError(Error):**
  - **"""Raised when the input value is too small"""**
  - **pass**
  - 
  - **class ValueTooLargeError(Error):**
  - **"""Raised when the input value is too large"""**
  - **pass**
  - 
  - **# our main program**
  - **# user guesses a number until he/she gets it right**
  - 
  - **# you need to guess this number**
  - **number = 10**
  - 
  - **while True:**
  - **try:**
  - **i_num = int(input("Enter a number: "))**
  - **if i_num < number:**
  - **raise ValueTooSmallError**
  - **elif i_num > number:**
  - **raise ValueTooLargeError**
  - **break**
  - **except ValueTooSmallError:**
  - **print("This value is too small, try again!")**
  - **print()**
  - **except ValueTooLargeError:**
  - **print("This value is too large, try again!")**
  - **print()**
  - 
  - **print("Congratulations! You guessed it correctly.")**

# Exception Handling

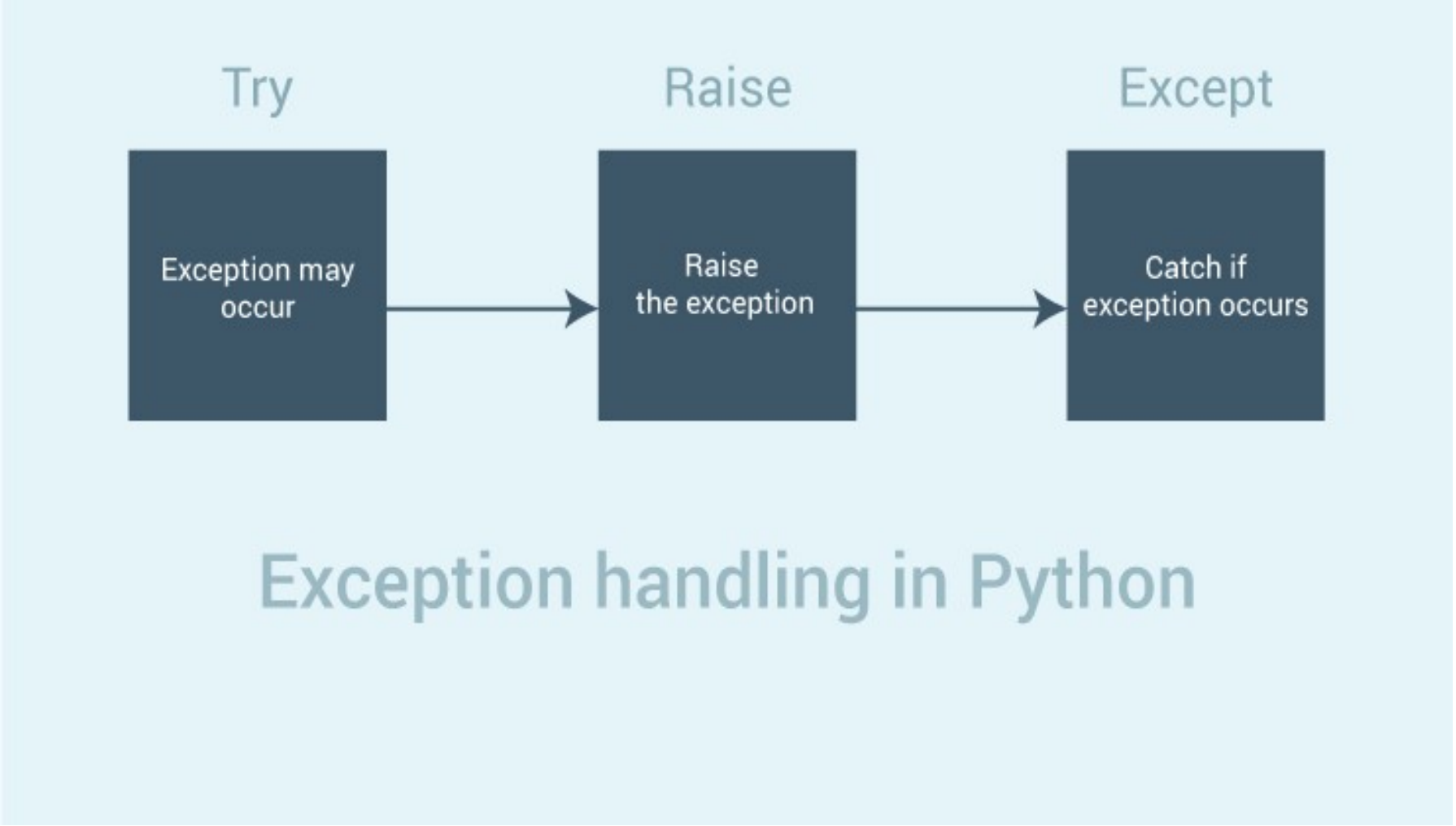
- Python Custom Exceptions
  - This program will ask the user to enter a number until they guess a stored number correctly.
  - Here is a sample run of this program.
    - Enter a number: 12
    - This value is too large, try again!
    -
    - Enter a number: 0
    - This value is too small, try again!
    -
    - Enter a number: 8
    - This value is too small, try again!
    -
    - Enter a number: 10
    - Congratulations! You guessed it correctly.

# Exception Handling

- Python Custom Exceptions
  - Here, we have defined a base class called Error.
  - The other two exceptions (ValueTooSmallError and ValueTooLargeError) that are actually raised by our program are derived from this class. This is the standard way to define user-defined exceptions in Python programming

# Exception Handling

- Python Exception Handling



# Exception Handling

- Python Exception Handling
  - Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.
  - When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.
  - For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.
  - If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

# Exception Handling

- Python Exception Handling
  - Catching Exceptions in Python
    - In Python, exceptions can be handled using a try statement.
    - A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Next entry.")
        print()
print("The reciprocal of",entry,"is",r)
```

# Exception Handling

- Python Exception Handling
  - Catching Specific Exceptions in Python
    - In the above example, we did not mention any exception in the except clause.
    - This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
    - We can specify which exceptions an except clause will catch.
    - A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.
    - We can use a tuple of values to specify multiple exceptions in an except clause.

# Exception Handling

- Python Exception Handling
  - Catching Specific Exceptions in Python
    - **try:**
    - **  # do something**
    - **  pass**
    - 
    - **except ValueError:**
    - **  # handle ValueError exception**
    - **  pass**
    - 
    - **except (TypeError, ZeroDivisionError):**
    - **  # handle multiple exceptions**
    - **  # TypeError and ZeroDivisionError**
    - **  pass**
    - 
    - **except:**
    - **  # handle all other exceptions**
    - **  pass**

# Exception Handling

- Python Exception Handling
  - try...finally
    - The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.
    - For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).
    - In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.

# Exception Handling

- Python Exception Handling
  - try...finally
    - try:
    - f = open("test.txt",encoding = 'utf-8')
    - # perform file operations
    - finally:
    - f.close()
  - This type of construct makes sure the file is closed even if an exception occurs.

# END of Core Python