**PYTHON 3**

1. Print
   1. print(“Hello world!”)
   2. print(10//4) #ans. 2 truncated operator
   3. print(10%3) #ans. 1 modulus operator
   4. print(4 \*\* 2) #ans. 1 power
   5. print(2 \*\* 3 \*\* 2) # the right-most \*\* operator gets done first!
   6. Print( type(30.12) ) # <class ‘float’>
2. Type Conversion Functions
   1. int() # it doesn’t round off
      1. print(**int(3.99)**) # ans. 3
      2. print(int(-3.99)) # ans. -3
   2. float()
   3. str()
3. Length
   1. print( len(“Hello”) ) # ans. 5
4. Input
   1. X= **input(“Enter the value: ”)** # always return string, use Typecasting
5. For Loop
   1. **for** \_ **in** range(10):

doSomething()

* 1. **for** loop\_var **in** [“Joe”, “Harry”, “Parth”, “Amy”]:

print(“Hello ”+ loop\_var)

* 1. **for** loop\_var **in** S:

print(“Hello ”+ loop\_var)

* 1. **for** loop\_var **in** myList:

print(“Hello ”+ loop\_var)

* 1. **for** loop\_var **in** myTuple:

print(“Hello ”+ loop\_var)

* 1. **for** loop\_var **in range(5):** # **range(5)** = [0, 1, 2, 3, **4**]

print(loop\_var)

* 1. range() is an iterable, it doesn’t return a list
  2. To get a list, typecast as **list( range(5) )**
  3. **range(1, 5)** = [1, 2, 3, **4**]

1. While Loop
   1. **while** condition**:**

do\_something

* 1. Use **break** or **continue** in the loops if required.

1. Random
   1. **import random**
   2. print( **random.random()** ) #can give any real numbers b/w 0&1
   3. print ( **random.randrange(1,7**) ) # int between [1,7)
   4. **from random import randrange, random**
   5. print( random() )
   6. print ( randrange(1,7) )
   7. **import random as rnd**
   8. print( rnd.random() )
   9. print ( rnd.randrange(1,7) )
2. Math
   1. **math.sqrt()**
3. Collection Data Types
   1. Strings (**Immutable**)
   2. List (**Mutable**)
   3. Tuples (**Immutable**)
   4. Dictionaries (**Mutable**)
4. String
   1. S= “Hello World”
   2. S= ‘Hello World’
   3. m = “””

This is a Multi-Line

String.

“””

* 1. m = ‘’’

This is a Multi-Line

String.

‘’’

* 1. print( **S[0]** ) # index possible
  2. **len (s)**  # no. of items
  3. print ( S[len(S) -1] ) = print ( S[-1] ) # negative indices possible
  4. print ( S[2:**6**] ) #Slicing # print S[2] to **S[5]**
  5. print ( S[:**6**] ) #Slicing # print S[0] to **S[5]**
  6. print ( S[**3**:] ) #Slicing # print S**[3]** to S[-1]
  7. **Slice always returns List**
  8. print( S + “abc” ) # Concatenation
  9. print(S \*3) # Repetition # Concatenation 3 times
  10. print( **S.count(“ri”)** ) # Counts no. of instances of “ri” in the given string S

# count item is **Case Sensitive**

* 1. print( **S.index(“ri”)** ) # returns index of **first** instance where 1st char of “ri” i.e. “r” #appears
  2. If index item is not present then Run Time Error.
  3. print( **S.split()** ) # Split # remove spaces in S and return list of Words

# [“Hello”, “World”]

* 1. print( **S.split(“o”)** ) # Split with “o” i.e. [“Hell”, “ W”, “rld”]
  2. print( **“o”.join([“Hell”, “ W”, ”rld”]**) #join # joins list of string with given string (here “o”)

# “Hello World”

* 1. A = “banana”

B = “banana”

# since string immutable # **IS operator checks if pointing to same object**

print( A **is** B ) # True

1. List
   1. myList = [“apple”, 5, “Banana”, 10]
   2. myList1 = list( range(4))
   3. print( **myList[0]** )
   4. **len( myList)**
   5. print (myList[ len(myList) -1] ) = print (myList [-1] )
   6. print ( myList[2:**6**] ) #Slicing # print myList[2] to myList**[5]**
   7. print ( myList [:**6**] ) #Slicing # print myList [0] to myList **[5]**
   8. print ( myList [**3**:] ) #Slicing # print myList **[3]** to myList[-1]
   9. **Slice always returns List**
   10. print( myList + [“abc”] ) # Concatenation
   11. print( **myList.append(“abc”)** ) # Concatenation
   12. print(myList \*3) # Repetition # Concatenation 3 times
   13. print( **myList .count(10)** ) # Counts no. of instances of 10 in the given list myList

# count item is **Case Sensitive**

* 1. print( **myList.index(“Banana”)** ) # returns index of **first** instance where

# “Banana”appears

* 1. If index item is not present then **Run Time Error**.
  2. **del** myList[1] # **delete** myList [1]
  3. **del** myList[1:3] # **delete** myList [1:3]
  4. A = [81, 82, 83]

B = [81, 82, 83]

# since List is mutable # different copies are made for each even if value same

print( A **is** B ) # False

print( A **==** B ) # True

# id operator gives back the id of the allocations

print( **id**(A) , **id**(B) ) # return diff. IDs

* 1. A = [81, 82, 83]

**B = A** # Aliasing

print(A **is** B) # **True**

# **B now point to the same object as A**

* 1. A = [81, 82, 83]

**B = A[ : ]** # Cloning

print(A **is** B) # **False**

* 1. A = [81, 82, 83]

B = A

* + 1. **B = B + [84]**

Print( A **is** B ) # **False**

# makes a new object entirely and reassigns to object B

* + 1. **B += [84] # avoid this with Lists**

Print( A **is** B ) # **True**

# modifies same object B

1. Tuple
   1. myTuple = (“apple”, 5, “Banana”, 10)
   2. myTuple = (500) # int not Tuple
   3. myTuple = (500,) # now it’s a Tuple
   4. myTuple = () # Empty Tuple
   5. myTuple = “apple”, 5, “Banana”, 10 **# python implicitly pack this to form a Tuple #OK** # No parenthesis
   6. fruit1, n1, fruit2, n2 = myTuple # **Unpacking**
   7. a, b, c, d = 1, 2, 3, 4 # **OK**
   8. print( **myTuple[0]** )
   9. **len( myTuple)**
   10. print (myTuple [ len(myTuple) -1] ) = print (myTuple [-1] )
   11. print ( myTuple [2:**6**] ) #Slicing # print myTuple [2] to myTuple **[5]**
   12. print ( myTuple [:**6**] ) #Slicing # print myTuple [0] to myTuple **[5]**
   13. print ( myTuple [**3**:] ) #Slicing # print myTuple **[3]** to myTuple [-1]
   14. **Slice always returns List**
   15. print( myTuple + “abc” ) # Concatenation
   16. print(myTuple \*3) # Repetition # Concatenation 3 times
   17. print( **myTuple.count(10)** ) # Counts no. of instances of 10 in

# the given tuple myTuple # count item is **Case Sensitive**

* 1. print( **myTuple.index(“Banana”)** ) # returns index of **first** instance where

# “Banana”appears

* 1. If index item is not present then **Run Time Error**.
  2. (a, b) = (b, a) # neat **swapping** of variables
  3. fruits = ['apple', 'pear', 'apricot', 'cherry', 'peach']

for item in **enumerate**( fruits ): # enumerate **returns tuple** of index

print(item[0], item[1]) # & value

**Output:**

0 apple

1 pear

2 apricot

3 cherry

4 peach

1. Strings II
   1. S.**upper()** # return uppercased # non-mutating
   2. S**.lower()** # return lowercased # non-mutating
   3. S.**count(‘l’)** # return no. of instances of ‘l’ in the string # non-mutating
   4. S.**strip()**  # removes any white spaces at the **beginning** & the **end**

# non-mutating

# Whitespaces in the **middle** are **not** removed

# not same as S.**split()**

* 1. S= S.**replace(‘o’, ‘\*’)** # replace any ‘o’s with ‘\*’ # non-mutating
  2. person = input( “Enter your name: “ )

print( “ Hello **{}**! ”**.format( person )** )

# replaces {} with entries in .format() # non-mutating

# It is important to pass arguments to the format method in the correct order, because they are matched positionally into the {} places for interpolation where there is more than one.

# Format strings can give further information inside the braces showing how to specially format data. In particular floats can be shown with a specific number of decimal places.

* 1. For two decimal places, put **:.2f** inside the braces for the monetary values:

# round-off

price = input( “Enter the price: “ )

print( “ Hello **{ :.2f }**! ”**.format( price )** ) # print float with 2 decimal places

1. List II
   1. myList**.append(5)** # mutation
   2. myList.**insert(1, 12)** # inserts 12 at myList[1] # mutation
   3. myList.**count(12)** # return no. of 12s in the list
   4. myList.**index(12)** # return index of first instance of no. 12
   5. myList.**reverse()** # reverse the whole list # mutation
   6. myList.**sort()** # ascending sort # mutation
   7. **del** myList[2] # deletes by **index** # mutation
   8. myList**.remove(12)** # deletes all **values** 12 # mutation
   9. myList.**pop()** # **return** & **delete** last value of list # mutation

It is important to remember that methods like append, sort, and reverse all return **None**

* 1. myList**.sort()** # Returns None # sort # mutation
  2. sList = **sorted(** myList **)** # Returns new sorted list # myList unchanged
  3. **sorted() can be applied to any data structure # preferred**
  4. sList = **sorted(** myList, **reverse = True )** # reverse sort
  5. def **absolute**(x): **# define a function & pass it as a key**

if x>=0: **# to sorted() # IMPORTANT**

return x # function passed as parameter

else:

return -x

sList = **sorted(** myList, **key = absolute )** # **sort according to absolute values**

* 1. sList = **sorted(** myList, reverse = True, **key = absolute )** # reverse absolute sort
  2. sList = sorted( myList, **key = lambda x: abs(x)** ) # pass **lambda function** as Key
  3. if ( **type(**x**) is list**) # to check if x is a list

1. Boolean
   1. Literal
      1. print( type(**False**) ) # <class ‘bool’>
      2. print( type(**True**) ) # <class ‘bool’>
      3. Boolean takes either True or False
   2. Comparison Operator
      1. print( 5==6 ) # False
      2. **==**
      3. **!=**
      4. **>**
      5. **>=**
      6. **<**
      7. **<=**
   3. Logical Operators (X &Y are Boolean values)
      1. X **and** Y # remember && from cpp
      2. X **or** Y # remember || from cpp
      3. **not** X # remember ! from cpp
   4. In/ Not In operator
      1. print( ‘p’ **in** ‘paper’) # True as ‘p’ is a substring of ‘paper’
      2. print( ‘p’ **in** ‘mango’) # False as ‘p’ not a substr of ‘mango’
      3. print( ‘’ **in** ‘apple’) # **True** as empty string
      4. print( ‘p’ **not in** ‘mango’) # True
      5. print( ‘pa’ **not in** ‘paper’) # False
      6. ‘apple’ **in** [‘apple’, ‘mango’] # True
2. Precedence Order
   1. Parentheses
   2. Exponent
   3. Multiplication, Division (\*, /, //, %)
   4. Subtraction, Addition
   5. Comparison Operator (==, !=, >=,<=,>,<)
   6. NOT Operator (not X)
   7. AND (and)
   8. OR (or)
3. Conditional Execution
   1. **If x%2 == 0**:

print( x, “is even”)

**else:**

print( x, “is odd”)

* 1. Chained Conditionals (ELIF)
     1. **if** x > y:

print(“greater”)

**elif** x <y:

print(“lower”)

**else**:

print(“equal”)

1. Mutation
   1. Changing the values either by making a copy or Modifying the original.
   2. Fruit = [“apple”, “banana”, “cherry”]
   3. Fruit[0] = “pear” # mutation
   4. Fruit[-1]=”orange” # mutation
   5. Fruit[1:3] = [ “pear”, ”orange”] # slice mutation
   6. **Fruit[1:3] = [ ]** # Deletes Fruit[1] to Fruit[2] # slice mutation
      1. len(Fruit) = 1 # since two items deleted
   7. **del** Fruit[1] # **delete** Fruit[1]
   8. **del** Fruit[1:3] # **deletes** Fruit[1:3]
   9. **Fruit[1:1] = [“pear”, “orange” ]** # Insert
      1. Print(Fruit)
      2. [“apple”, “pear”, “orange”, “banana”, “cherry”]
   10. greeting = “Hello World!”
       1. greeting[0] = ‘J’ # ERROR # Strings immutable
       2. new\_greeting = ‘J’ + greetings[1: ] # **Allowed**

# Jello World!

* 1. A = “banana”

B = “banana”

# since string immutable # **IS operator checks if pointing to same object**

print( A **is** B ) # True

* 1. A = [81, 82, 83]

B = [81, 82, 83]

# since List is mutable # different copies are made for each even if value same

print( A **is** B ) # False

print( A **==** B ) # True

# id operator gives back the id of the allocations

print( **id**(A) , **id**(B) ) # return diff. IDs

1. **Aliasing** ( Important )
   1. A = [81, 82, 83]

B = A

print(A **is** B) # **True**

# **B now point to the same object as A**

* 1. A = [81,82,83]

B = [81,82,83]

print(A **is** B) # False

B = A

print(A == B)

print(A **is** B) # True

**B[0] = 5**

**print(A)**

Output:

False  
True  
True  
[**5**, 82, 83]

* 1. A = [81, 82, 83]

B = A[ : ] # Cloning

print(A is B) # **False**

* 1. A = [81, 82, 83]

B = A

* + 1. **B = B + [84]**

Print( A **is** B ) # **False**

# makes a new object entirely and reassigns to object B

* + 1. **B += [84]**

Print( A **is** B ) # **True**

# modifies same object B

1. Working with Data Files
   1. File1 = **open(“Olympics.txt”, “r")** # open a file # read only
   2. File1 = **open(“Olympics.txt”, “w”)** #also open a new file # open a file # writing
   3. File1.**close()** # File use complete
   4. Contents = File1**.read()** # **entire content** in olympics.txt as a **single string**
      1. Print( Contents[:100] )
   5. Contents\_List = File1.**readlines()** # returns list of **each lines of the entire content**

# each line also **include ‘\n’ at the end**

* + 1. print( Contents\_List[:10] ) # Print first 10 lines of the files as a list
    2. **for** line **in** Contents\_List[:10]:

print(line) # Print first 10 lines exactly as in the file

* + 1. **for** line **in** Contents\_List[:10]:

print( **line.strip()** ) # Use .strip() to **remove ‘\n’** at each line end

* 1. **for** line **in** File1: # preferred when large data compared to readlines()

print( line.strip() ) # Use file variable directly to traverse over the whole file

# return each lines

* 1. **for** line **in** File1[:10]: # **Error**  # only File1 allowed # only whole file
  2. File1.**readline()** # return next line
  3. File1.**readlines()**  # return list of lines
  4. File1.**read(10)** # read first 10 chars
  5. File1.**write(“Something!!”)** # Add string to the end of the file
  6. File1.**write ( str() )** # the argument has to be string
  7. File1.**write( str() + “\n” )**  # add new line for formatting
  8. Specifying File Path
     1. Relative
        1. You have to use ‘ **../** ’. This implies go to parent directory
        2. **open(‘../myData/ file2.txt’ , ‘r’)**
     2. Absolute
        1. **open(‘/e/folder1/myData/file2.txt’, ‘r’ )**
        2. Absolute path starts with **‘/’**
  9. File1 = **open(“Olympics.txt”, “r")**
  10. **with** open(“Olympics.txt”, “r") **as** File1: # needs **indentation** after this line

# with/ as is same as previous open but **doesn’t require** a File1.**close**()

# with/ as is the **preferred way** to open a file

1. CSV Format (Comma Separated Values)
   1. Conventions
      1. Comma Separated “,”
      2. First line defines column names
      3. All other rows or lines follows the same above structure
   2. Text = [ (1,2,3) , (4,5,6) , (7,8,9) ] # csv format for writing on file
   3. If there’s a comma used in the data then use “” to enclose each data
2. Dictionaries
   1. **Unordered** Data Collection unlike String, List, Tuple
   2. Stores Key, Value pair
   3. nameAge = **{}** **# declare a dictionary by { }**

nameAge[ ‘Adam’ ] = ‘24’ # key: ‘Adam’ Value: ‘24’

nameAge[ ‘Steve ] = ‘31’

nameAge[ ‘Java ] = ‘14’

or you can directly declare:

nameAge = {‘Adam’ **:**  ’24 ’ **,**  ‘Steve’ **:** ’31’ **,**  ‘Java’ **:** ‘14’}

* 1. **print(nameAge)**

>> {‘Adam’ **:**  ’24 ’ **,**  ‘Steve’ **:** ’31’ **,**  ‘Java’ **:** ‘14’}

* 1. print( **nameAge[‘Steve’]** )

>> 31

* 1. **nameAge[‘Steve’] = ‘35’**  # **mutable**
  2. **del** nameAge[‘Steve’] # deletes key: Steve & value: 31
  3. **len(**nameAge**)** # return no. of key value pairs
  4. nameAge**.keys()** # return all Keys (not List but **Iterable**)
  5. **for** key **in** nameAge**.keys():**

print( **nameAge[key]** )

* 1. **for** key **in** **nameAge:**

print( **nameAge[key]** ) # same as previous

* 1. **for** **k, v in** **nameAge.items():**

**print( k, v )**  **# Tuple Unpacking**

* 1. key\_list = list( nameAge.keys() ) # Typecast to list of keys
  2. nameAge**.values()** # return all values
  3. list( nameAge**.values() )** # return list of values
  4. nameAge**.items()** # return key/value pair as Tuple
  5. list( nameAge**.items()** ) # return list of key/value Tuples
  6. **print(** ‘Steve’ **in** nameAge **)** **# only for keys** # return Boolean TRUE/ FALSE
  7. print( nameAge**.get(** ‘Steve’ **)** ) # return value of key in argument
  8. print( nameAge.**get(** ‘Mark’ **)** ) # returns **“None”** if key doesn’t exist
  9. print( nameAge**[** ‘Mark’ **]** ) # Run Time **Error**
  10. print( nameAge.**get(** ‘Mark’ , 0 **)**) # returns **0** if key doesn’t exist
  11. nameAge\_alias = nameAge # alias points the same object here

print(nameAge\_alias **is** nameAge) # returns True

* 1. nameAge\_copy = nameAge**.copy()** # makes a copy instead of alias
  2. **sorted** **(** nameAge.keys() **)** # **sort** Dictionary **Keys**
  3. **sorted** **(** nameAge **)** # **sort** Dictionary **Keys # same**
  4. **sorted (nameAge.keys(), key = lambda x: nameAge[x] )** # **sort** Dictionary **Values**
  5. sorted (nameAge.keys(), reverse = True, key = lambda x: nameAge[x] )

# **reverse sort** Dictionary **Values**

* 1. Key of the dictionary has to be a Immutable object (So no List, Dictionary as key)

1. Functions
   1. **def** name**( parameters ):**

statements

* 1. **def** name**( parameters ):**

statements

**return** something

* 1. Returns **None** if no return specified.
  2. **def** name**( parameters ):**

**global any\_variable** # If you want global variable access

statements

**return** something

* 1. **def** name**( c ):**

a = b + c # if b is global then **OK**

**return** a

b = 3

name (5)

* 1. **def** name**( c ):**

a = b + c # **ERROR** since 1st local ? If yes:

b = 2 # defined later in function

**return** something # no global access here

b = 3

name (5)

* 1. **IMP:** Variable can be local but objects like List, Dictionary (Mutable) are passed as global scope. Hence the **function will mutate the original list or Dictionary**.
  2. **How To avoid:**
     1. New\_list = func ( **list( original\_list)** )
     2. New\_Dict = func ( **dict( original\_Dict)** )
     3. The function **list() & dict() returns a new copy of the argument**. Pass this as a parameter to the function to avoid mutation.
  3. **def** name**( b, c ): #** only one item can be returned

a = b + c

**return** (a,b,c) **# Use Tuple to return more than 1 values**

* 1. **def** name**( b, c ):**

a = b + c #return Tuple

**return** a,b,c **# Implicit packing** into a Tuple **# OK**

**x, y, z = name (5,4)** # **Tuple Unpacking**

* 1. **return [a, b, c]** is not the preferred method because it **returns a, b and c in a mutable list rather than a tuple** which is more efficient. But it is workable.
  2. **def** name**( b, c ):**

a = b + c

**return** a,b,c

**tup = 5,4**

**x, y, z =** name(**\*tup**) #  **\* => automatic Unpacking**

* 1. **def** name**( b, c ):**

a = b + c

**return** a,b,c

**tup = 5,4**

**x, y, z =** name(**tup**) #  **ERROR**

* 1. initial = 7 # **IMP**

def f(x, y =3, z=initial): **# Optional Parameter**

print("x, y, z, are: " + str(x) + ", " + str(y) + ", " + str(z))

initial = 10

f(2)

**Output:**

x, y, z, are: 2, 3, 7

* 1. def f (a, L=[] ): #**IMP**

L.append(a) # **Mutation**

return L # L optional

print( f(1) )

print( f(2) )

print( f(3) )

print( f(4, ["Hello"]) )

print( f(5, ["Hello"]) )

**Output:**

[1]

[1, 2]

[1, 2, 3] # Local L mutable

['Hello', 4]

['Hello', 5]

* 1. initial = 7 # **IMP**

**def f( x, y =3, z = initial):** **# Optional Parameter**

print("x, y, z, are: " + str(x) + ", " + str(y) + ", " + str(z))

initial = 10

**f(2, z = 8) # Keyword Parameter**

**# f(2, , 8) will give an error**

**# f(x=2, z=8) OK**

**# f(z=8, x=2) OK No order required**

**Output:**

x, y, z, are: 2, 3, 8

* 1. def func (args):

return ret\_val

* 1. **Lambda Function**

**func = lambda args: ret\_val** # Other way to write above function

**lambda args: ret\_val** # Anonymous function # **OK**

* 1. **L = [**fun1, fun2, lambda x: abs(x) **]** # You can create list of functions too

**Print ( L[0](parameter) )**

1. Advance Sorting (Second Sorting or Sorting on more than 1 property)
   1. Make **Tuple** of all the properties & sort these Tuples.
   2. Tuple sorting: Sort according to 1st element.
   3. Tie break: If 1st same, sort according to 2nd and so on
   4. Utilize above for advance sorting
   5. fruits = ['peach', 'kiwi', 'apple', 'blueberry', 'papaya', 'mango', 'pear']

**# sort according to length and letter**

new\_order = **sorted(**fruits, **key=lambda fruit\_name: (len(fruit\_name), fruit\_name))**

**output:**

kiwi

pear

apple

mango

peach

papaya

blueberry

* 1. fruits = ['peach', 'kiwi', 'apple', 'blueberry', 'papaya', 'mango', 'pear']

**# reverse sort according to length and letter**

new\_order = **sorted(**fruits, **key=lambda fruit\_name: (len(fruit\_name), fruit\_name, reverse = True))**

**output:**

blueberry

papaya

peach

mango

apple

pear

kiwi

* 1. fruits = ['peach', 'kiwi', 'apple', 'blueberry', 'papaya', 'mango', 'pear']

**# reverse sort according to length but ascending letter**

new\_order = **sorted(**fruits, **key=lambda fruit\_name: (-len(fruit\_name), fruit\_name))**

**output:**

blueberry

papaya

apple # alphabetical order retained

mango

peach

kiwi

pear

1. JSON ( Java Script Object Notation )
   1. Standard format to share Nested Lists & Dictionaries.
   2. Difference from Python:
      1. Null (instead of None)
      2. true/ false (instead if True/ False)
   3. **import** **json**
   4. **json.loads(**any\_string\_input**)** # takes string (Json) as input & return either

# Dictionary / List depending on string

* 1. **json.dumps(**any\_List\_or\_Dictionary**)** # opposite of loads()

# takes List or Dictionary and return (JSON)

# string

* 1. json**.dumps(** obj, **sort\_keys = True**, **indent =2**) # if dictionary input, sort\_key

# available

* 1. **Use** [**jsoneditoronline.org**](jsoneditoronline.org) **to understand JSON data in levels.**

1. Shallow & Deep Copy (Important)
   1. When there are multi-level nesting in a list & you try to make a copy of it by:
   2. New\_list = original\_list [:] **# it will make 1st level true copy only # rest levels are aliasing # This is called a shallow copy**
   3. x=4

y=2

z=3

myList = [[x,y,z],0,0]

print(myList)

x=6

print(myList)

**Output:**

[[4, 2, 3], 0, 0]

[[4, 2, 3], 0, 0]

* 1. original = [['dogs', 'puppies'], ['cats', "kittens"]]

copied\_version = original[:]

print(copied\_version)

print(copied\_version is original)

print(copied\_version == original)

original[0].append(["canines"])

print(original)

print("-------- Now look at the copied version -----------")

print(copied\_version)

**Output:**

[['dogs', 'puppies'], ['cats', 'kittens']]

[['dogs', 'puppies', ['canines']], ['cats', 'kittens']]

-------- Now look at the copied version -----------

[['dogs', 'puppies'], ['cats', 'kittens']]

* 1. original = [['dogs', 'puppies'], ['cats', "kittens"]]

copied\_outer\_list = []

for inner\_list in original:

copied\_inner\_list = inner\_list[:]

copied\_outer\_list.append(copied\_inner\_list)

print(copied\_outer\_list)

original[0].append(["canines"])

print(original)

print("-------- Now look at the copied version -----------")

print(copied\_outer\_list)

**Output:**

[['dogs', 'puppies'], ['cats', 'kittens']]

[['dogs', 'puppies', ['canines']], ['cats', 'kittens']]

-------- Now look at the copied version -----------

[['dogs', 'puppies'], ['cats', 'kittens']]

* 1. **Important**

**import copy**

original = [['canines', ['dogs', 'puppies']], ['felines', ['cats', 'kittens']]]

shallow\_copy\_version = original[:]

deeply\_copied\_version = **copy.deepcopy(**original**)**

original.append("Hi there")

original[0].append(["marsupials"])

print("-------- Original -----------")

print(original)

print("-------- deep copy -----------")

print(deeply\_copied\_version)

print("-------- shallow copy -----------")

print(shallow\_copy\_version)

**Output:**

-------- Original -----------

[['canines', ['dogs', 'puppies'], [**'marsupials']],** ['felines', ['cats', 'kittens']], **'Hi there'**]

-------- deep copy -----------

[['canines', ['dogs', 'puppies']], ['felines', ['cats', 'kittens']]]

-------- shallow copy -----------

[['canines', ['dogs', 'puppies'], [**'marsupials']],** ['felines', ['cats', 'kittens']]]

1. MAP
   1. Function that changes the list items and return new list.
   2. Function (modifying) as 1st parameter and Sequence as 2nd parameter.
   3. new\_list = **map(** **lambda x: 2\*x** , **myList** **)**
   4. new\_list = map( double, myList ) # define a function double()
2. Filter
   1. Start with some items and end up with fewer items.
   2. Gives back an iterable object.
   3. new\_seq = **filter(** **lambda x: x%2==0 , myList )** # only choose even numbers
   4. new\_seq = list( new\_seq )
   5. lst2 = **filter( lambda w: “o” in w , myList** **)** # choose words with letter ‘o’

NOTES

1. Some general errors:
   1. Syntax Error
   2. Logic Error
   3. Compilation Error
   4. Run time Error
   5. Resource Error
   6. Interface Error
2. Python is an Interpreted Language.
3. Workflow:
   1. Compiler
      1. Source Code
      2. Compiler
      3. Object Code or the executable
      4. Executor
      5. Output
   2. Interpreters
      1. Source Code
      2. Interpreter
      3. Output
4. Variables
   1. Can only start with a letter
   2. Can only contain letters and numbers.
   3. Underscore is also allowed.
   4. Variable names can never contain spaces.
   5. It should not be a Python Keyword (Ex: class)
5. The input function returns a string value
6. Functions imported as part of a module live in their own **namespace**. A namespace is simply a space within which all names are distinct from each other. The same name can be reused in different namespaces but two objects can’t have the same name within a single namespace.
7. Python Libraries <https://docs.python.org/3.6/library/index.html>
   1. <https://docs.python.org/3.6/py-modindex.html>
8. Don’t overwrite standard library modules!
9. It is important to note that random number generators are based on a deterministic algorithm — repeatable and predictable. So they’re called **pseudo-random generators** — they are not genuinely random. They start with a seed value. Each time you ask for another random number, you’ll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated. The good news is that each time you run your program, the seed value is likely to be different, meaning that even though the random numbers are being created algorithmically, you will likely get random behaviour each time you execute.
10. \* operator in repetition of concatenation follows the same preference rule as of Multiplication
11. Mutation operations are often called **Destructive** operations as they change the values.
12. Python has the notion of a **context manager** that automates the process of doing common operations at the start of some task, as well as automating certain operations at the end of some task. For reading and writing a file, the normal operation is to open the file and assign it to a variable. At the end of working with a file the common operation is to make sure that file is closed.