**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. 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. 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. print( myTuple[0] )
   6. len( myTuple)
   7. print (myTuple [ len(myTuple) -1] ) = print (myTuple [-1] )
   8. print ( myTuple [2:**6**] ) #Slicing # print myTuple [2] to myTuple **[5]**
   9. print ( myTuple [:**6**] ) #Slicing # print myTuple [0] to myTuple **[5]**
   10. print ( myTuple [**3**:] ) #Slicing # print myTuple **[3]** to myTuple [-1]
   11. **Slice always returns List**
   12. print( myTuple + “abc” ) # Concatenation
   13. print(myTuple \*3) # Repetition # Concatenation 3 times
   14. 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**.

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.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. 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

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.