Dharmsinh Desai University, Nadiad
Faculty of technology,
Department of Computer Engineering
Subject : Software Project
Lab Manual

Lab2 : Data structures and Functions in Python.

Aim : To learn Data Structures in Python.

Requirements : Python, IDE(IDLE, Jupyter Notebook)

Python programming supports following data structures.

- Lists
- Tuples
- Dictionaries
- Strings
- Sets

❖ **List:** In Python programming, a list is created by placing all the items (elements) inside a square bracket [ ], separated by commas. Some examples are as below:

```
In [1]: """Python Lab 2"""
        my_list = []
        print(my_list)
        # list of integers
        my_list = [1, 2, 3]
        print(my_list)
        # list with mixed datatypes
        my_list = [1, "Hello", 3.4]
        print(my_list)
```

```
[]
[1, 2, 3]
[1, 'Hello', 3.4]
```

➢ 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. Python also allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

Example:

```
In [6]: l=['h','e','l','l','o','w','o','r','l','d']
        print(l)
        print(l[2])
        print(l[-4])
```

```
['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
l
o
```

- We can access a range of items in a list by using the slicing operator

Example:

```
In [7]: print(l[2:6])
        ['l', 'l', 'o', 'w']

In [8]: l[:4]
Out[8]: ['h', 'e', 'l', 'l']

In [9]: l[6:]
Out[9]: ['o', 'r', 'l', 'd']
```

- Apart from this list provides some other methods to manipulate list. Those methods are as below

- **l.append(*x*) :**Add an item to the end of the list. Equivalent to a[len(a):] = [x].

- **l.extend(*iterable*) :** Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.
- **l.insert(*i, x*) :** Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).
- **l.remove(*x*) :**Remove the first item from the list whose value is *x*. It is an error if there is no such item.
- **l.pop([*i*])** :Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
- **l.clear()** :Remove all items from the list. Equivalent to del a[:].
- **l.index(*x*[, *start*[, *end*]])** : Return zero-based index in the list of the first item whose value is *x*. Raises a ValueError if there is no such item. The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.
- **l.count(*x*) :** Return the number of times *x* appears in the list.
- **l.sort()** : Sort the items of the list in place.
- **l.reverse()** : Reverse the elements of the list in place.
- **l.copy()** : Return a shallow copy of the list. Equivalent to a[:].

List Comprehansion:

```
In [123]: #list comprehansion in python.
          #list comprehension always returns a result list
          a=[i for i in range(1,10)]
          a

Out[123]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Difference between append and extend:

```
In [124]: a.append(16)
          print(a)
          a.extend([17])
          print(a)
          a.extend([18,19])
          print(a)
          a.append([20,21])
          print(a)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 16]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 17]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 17, 18, 19]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 17, 18, 19, [20, 21]]
```

```
In [125]: a.append(22,23)
          print(a)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-125-2dad347c1dea> in <module>()
----> 1 a.append(22,23)
      2 print(a)

TypeError: append() takes exactly one argument (2 given)
```

- **Tuple:** A tuple consists of a number of values separated by commas either with round paranthesis or without it.

  Note: Tuple is immutable. So we cannot modify any data using index.

  Example:

```
In [11]: t=1,3,"hello",4,3.5
         t

Out[11]: (1, 3, 'hello', 4, 3.5)

In [12]: t[1]

Out[12]: 3

In [13]: t[3]=4.0

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-13-a32d37638c9a> in <module>()
----> 1 t[3]=4.0

TypeError: 'tuple' object does not support item assignment
```

- **Sets:** A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.
  - Curly braces or the set() function can be used to create sets.
    Note: to create an empty set you have to use set(), not {};

Example

```
In [14]: s={1,2,3,4,5,6,7}
         s

Out[14]: {1, 2, 3, 4, 5, 6, 7}

In [15]: s1=set("hello")
         s1

Out[15]: {'e', 'h', 'l', 'o'}

In [20]: s2={'h','e','l','l','o'}
         s2

Out[20]: {'e', 'h', 'l', 'o'}
```

- **Dictionary:** Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.

You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

- It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

Example:

```
In [21]: tel={'abc':123, 'xyz':456}
         tel

Out[21]: {'abc': 123, 'xyz': 456}

In [22]: tel['abc']

Out[22]: 123

In [23]: 'abc' in tel

Out[23]: True

In [24]: 'abc' not in tel

Out[24]: False
```

- The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Example:

```
In [25]: d={"abc":123, "def":345, "xyz":67}
         d
Out[25]: {'abc': 123, 'def': 345, 'xyz': 67}

In [26]: del d['abc']

In [27]: d
Out[27]: {'def': 345, 'xyz': 67}

In [28]: del d

In [29]: d
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
 last)
<ipython-input-29-e983f374794d> in <module>()
----> 1 d

NameError: name 'd' is not defined
```

❖ Converting Structures into other- mix and merge:

```
In [30]: s='cat'
         print(s)
         i=iter(s)
         l=[]
         l.extend(i);
         print(l)

cat
['c', 'a', 't']
```

❖ Shallow copy vs Deep Copy:

- Python defines a module which allows to deep copy or shallow copy mutable object using the inbuilt functions present in the module"**copy**".
- Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other.
- Ref.:(1) https://www.programiz.com/python-programming/shallow-deep-copy
  (2) https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/

❖ Exercise:[Based on data Structures]

NOTE: Implement following programs without using user defined function.

[1]. Write a program that stores following names in a list.Also write instruction to iteratively print names from the list.

‘ramu’, ‘shyamu’, ‘kanu’, ‘manu’, ‘ramu’, ‘radha’, ‘manu’.

[2]. Write a program to remove duplicates from the list (created in Ex:(1)) and print unique names.

[3]. Write a program to create a record of student details. The record maintains following information < student_id, name, age, percentage>.

Record1 : 1, ‘ram’, ‘18’, 65%

Record 2 : 2, ‘shyam’, 17’, 70%

add following records to the ‘student_record’ created in (3)

Record 3 : 3, ‘Radha’, ‘17’, 75%

[4]. Write a program to concat student records (created in (3)) of different semesters.

[5]. Write a program to print record of first 5 students.

[6]. Write a program to delete record of a specific student (e.g. ramu).

[7]. Write a program to combine following dictionaries by averaging values for common keys.

D1 = {‘ram’:60, ‘shyam’:70, ‘radha’:70}

D2 = {‘ram’:70, ‘shyam’:80, ‘gopi’:60}

output : {‘ram’:65, ‘shyam’:75, ‘radha’:70, ‘gopi’:60}

[8]. Write a program to sort student record by values (e.g. name).

[9]. Write a program to print all unique values in a dictionary.

Sample Data : [{"V":"S001"}, {"V": "S002"}, {"VI": "S001"}, {"VI": "S005"}, {"VII":"S005"}, {"V":"S009"},{"VIII":"S007"}]

Output : Unique Values: {'S005', 'S002', 'S007', 'S001', 'S009'}

[10]. Following are the lists of students studying Maths, Physics and Chemistry subjects,

Maths = {1, 2, 3, 5, 7, 9}

Physics = { 2, 4, 6, 9}

Chemistry = {1, 3, 5, 9}

Display the numbers which are studying both maths & physics, physics & chemistry, maths&chemistry and all three. Also, display the numbers which are studying exactly one of the three subjects.

❖ **Function**

Function is a piece of code written to carry out a specified task. To carry out that specific task, the function might or might not need multiple inputs. When the task is carried out, the function can or can not return one or more values.

❖ **Types:**

- Built-in functions, such as min() to get the minimum value, print() to print an object to the terminal.
- User-Defined Functions (UDFs), which are functions that users create to help them out.
- Anonymous functions, which are also called lambda functions because they are not declared with the standard def keyword.

1. User-Defined Functions: Here are simple rules to define a function in Python.

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax :

```
def functionname( Parameters ):
    "function_doString"
    function_body
    return [expression]
```

Example:

```
def printLine(str) :
        "function to print line"
        print(str)
        return

printLine('hello')
```

❖ **Pass by reference vs value:** All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Example:

```
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print ("Values inside the function: ", mylist)
   return


mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

- Output of this code will be as below:

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

❖ **Function Arguments:** There are four different types of argument in function as below:

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

1) Required Arguments: Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

Example:

```
def fun1(name,age):
        print("Name :",name)
        print("Age :",age)
        return

fun1('XYZ',21)
```

2) Keyword Arguments: Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Example:

```
def fun1(name,age):
        print("Name :",name)
        print("Age :",age)
        return

fun1(age=21,name='xyz')
```

3) Default Argument: A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments.

```
def fun1(name,age=35):
        print("Name :",name)
        print("Age :",age)
        return

fun1(age=21,name='xyz')
fun1('xyz')
```

4) Variable length Argument: You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax:

```
def functionname([formal_args,] *var_args_tuple ):
        "function_docstring"
        function_suite
        return [expression]
```

Example:

```
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print("Output is: ")
   print(arg1)
   for var in vartuple:
      print(var)
   return;
printinfo(1)
printinfo(1,2,3,4,5)
```

2. Anonymous Functions : These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Syntax :

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

Example:

```
sum = lambda arg1, arg2: arg1 + arg2;

print("Value of total : ", sum( 10, 20 ))
print("Value of total : ", sum( 20, 20 ))
```

Exercise[Based on Function]:

1. Check the output of following code and justify your answer:

```
x = 50
def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)
func(x)
print('x is now', x)
```

2. Write a function to print largest number from provided 3 values as parameter.
3. Write a function to print prime numbers between provided range.
4. Write a function to swap two values provided as arguments.
5. Write a function to compute simple interest using provided values as arguments if rate of interest is not provided consider it as 3.5%.
6. Write a function to compute the sum of data of variable length.