CMSC 245 – Principles of Programming Languages
Lab#4: Python: Lists, Slicing, Tuples, Functions, Command Line Arguments

In this lab we will learn about lists, slices, and functions in Python. We recommend that you try all the features presented in this lab. As you proceed, whenever you have any questions, go ahead and try them out in code to get your answers. This is the best way to learn Python.

Lists (aka Arrays)

In C, C++, or Java, an array is an indexed sequence of elements of the same type. Python's primary indexed structure is a list.

```
a = list()
data = [10, 20, 30, 40, 50]
primes := [2, 3, 5, 7, 11, 13]
```

Here is a short synopsis of the above definitions:

- **a** is an empty list (i.e. **[]**).
- **data** is a list of 5 integers containing the values supplied.
- **primes** is a list of 5 integers, all prime numbers.
- In all cases indexing in lists begins with 0. To access the elements of a list, you write **a[0]**, **a[1]**, …, etc.
- A list is created at the time of its first use. And, its size can be altered (see below). To get the number of elements in a list, do:

  ```
  len(a)          # returns 0
  len(data)       # returns 5
  len(primes)     # returns 6
  ```

- Lists can be assigned using the assignment (**=**) operator:

  ```
  p = primes
  ```

  p[] will point to **primes[]**. Thus **p[1]** and **primes[1]** refer to the same element (3). Altering one will also alter the other (i.e. **p** and **primes** are aliases):

  ```
  p[1] = 17
  # Now, p = [2, 17, 5, 7, 11] and primes = [2, 17, 5, 7, 11]
  ```

- To create a new list, you can do this:

  ```
  p = list(primes)
  ```

  A new list, **p** is created and initialized with the elements in **primes**. Now, **p** and **primes** are two different list objects, containing the same elements. And, changing an element in **p** does not affect **primes**. Since they are not aliases.

- In Python a list can contain elements of any type (including other lists):

  ```
  course = ['CMSC', 325, "Principles of Programming Languages"]
  m = [13, ['a', 'b', 'c'], 'Bryn Mawr', 3.141592654]
  ```

  Thus **m[1]** will be **['a', 'b', 'c']** and **m[1][1]** will be **'b'**.

- To iterate through all elements of a list:

  ```
  sum = 0
  for i in range(len(data)):
      sum = sum + data[i]
  ```

  The expression **range(n)** returns a list **[0, 1, 2, n-1]**. Or, you can use the loop:

  ```
  sum = 0
  for x in data:
      sum = sum + x
  ```

  Above, **x** will take on values in data at each iteration (i.e. 10, 20, 30, 40, 50).

Slicing

Using slicing you can select a range of values in a list.

```
primes := [2, 3, 5, 7, 11, 13]
a = primes[0:3]       # a is now the list [2 3 5]
```

Slicing creates a new list object, **a** and copies the indexed entries from **primes** into a. You can specify any start and end index (above we have 0 as start and 3 as end). The resulting slice is taken from start..end-1.

Start and end indexes can be omitted:

```
a = primes[:3]        # a is now the list [2, 3, 5]
b = primes[3:]        # b is now the list [7, 11, 13]
```

To copy the entire list, you can also do:

```
p = primes[:]         # p is a copy of primes
```

Tuples

Python also has tuples: a group of related values. For example ('cat', 'NOUN') to designate a cat is a noun. Or, (3, 4, 5) or (5, 12, 13) which are Pythagorean triplets ($3^2 + 4^2 = 5^2$):

```
part_of_speech = ('cat', 'NOUN')
triplet1 = (3, 4, 5)
triplet2 = (5, 12, 13)
```

Values in tuples can be any Python value (including a list!). Values in a tuple can be accessed using list-style indexing:

```
word = part_of_speech[0]
tag = part_of_speech[1]

a = triplet1[0]
b = triplet2[1]
c = triplet3[3]
```

Since a list can have any value in it, tuples can also be elements in a list:

```
triplets = [(3, 4, 5), (5, 12, 13), (8, 15, 17), (20, 21, 29)]
```

Functions

Functions in Python require you to name all the parameters. However, as required in C, C++, and Java, you do not need to specify the types of parameters and return value. Here is a simple function:

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

The syntax is:

def *function-name*(*parameters*):
  function-body

Another Example: To compute the maximum value in a list:

```
def max(a):
   m = a[0]
   for x in a[1:]:
       if a[i] > m {
           m = a[i]
   return m
```

The above is for illustration purposes only (to learn how to define a function and to show you how you can use slicing). Python has a built-in function **`max()`** that you can use:

```
m = max(a)
```

Variable Number of Arguments to Functions

Python allows you to write functions that may take an unknown number of parameters. For example, to compute the largest of one or more numbers:

```
a = max(x)
b = max(x, y)
c = max(w, x, y, z)
```

Such functions are written as shown below:

```
def max(*args):
      m = args[0]
      for x in args[1:]:
            if x > m:
                  m = x
      return m
```

Above, the parameter ***`args`** could be one or more integers. We can then traverse **`args`** as if it were a tuple.

Exercise: Define the function above and try it with the following:

```
a = max(5)
b = max(5, 9)
c = max(5, 9, 19, 3, 1)
```

Default Function Arguments

Python allows you to provide default values for function parameters. For example,

```
def power(x, y=2):
     '''power(x, y): returns x^y. By default y=2,
        unless specified otherwise.'''
     …etc…
```

Above, **`power()`** is defined to take two arguments: **x**, and **y**. However, a default value for **y** is provided (**y=2**). Thus, we can use **`power()`** as shown below:

```
a = power(10)          # Returns 10^2
b = power(10, 3)       # Returns 10^3
```

Note: Only the latter arguments can be defined with default values.

Multiple Return Values

Functions can also return more than one return value. For example, if we wanted to find out the value and index of the largest value in a list:

```
def max(a):
     m = a[0]
     i = 0
     for j in range(1, len(a)):
          if a[j] > m:
               m = a[j]
               i = j
     return (i, m)
```

You can now call the function `max()` as shown below:

```
index, value := max(primes)
```

Many functions in Python take advantage of this feature.

Command Line Arguments

Like in C, C++, and Java, Python, programs accept command line arguments. The **`sys`** module provides the list **`sys.argv[]`** that contains one string for each command line argument. **`Sys.argv[0]`** is the command itself.

The program below prints out all the command line arguments:

```
# Save in file: args.py
import sys
print('Arguments:', sys.argv)
```

Run it from a command line:

```
$ python args.py Deepak 31 43
Arguments: ['args.py', 'Deepak', '31', '43']
```

Exercise (Assignment 4)

1. Collecting Cards in Candy: A candy company wishes to run a promotion in the next quarter. Every candy wrapper will contain a picture of a player from the Women's National Soccer Team. There are 28 players on the roster, plus 7 coaches, making a total of 35 pictures. Pictures will be randomly inserted in each candy wrapper. The company hopes that children buying their candy will engage in collecting all 35 pictures and placing them in a picture album (sold separately by the company). Each picture will carry a unique number (from 0..34).

Before running with the idea the company wants to examine how this promotion might improve sales of its candy.

Write a program in Python that performs a simulation to determine the number of candies a child would have to purchase in order to collect all 35 cards. Your program should input the number of picture (say 35) on the command line argument:

```
$ python collector.py 35
XXX
```

The program will print out the expected number of candies (**XXX**) that children will purchase to collect all 35 cards. Here is a possible design:

1. Write a function **`collect(c)`** that will perform a simulation of collecting **`c`** cards. It will return the total number of candies bought.
2. Once written, you can use **`collect()`** to perform several trials and keep track of the number of candies bought:

    ```
    n = Number of cards to collect
    trials = 1000000
    sum = 0
    do trials times:
      sum = sum + collect(n)
    output sum/trials as the average number of candies to buy in order to collect n cards
    ```

    As the number of trials increases, the data will start to converge. Run your program to perform 1000,000 trials.
3. Run the program with 1000,000 trials to compute the number of candies required to be bought to collect 35, 100 pictures, and 500 pictures.