National University of Computer and Emerging Sciences

Lab Manual AL2002-
Artificial Intelligence Lab

Department of Computer Science
FAST-NU, Lahore, Pakistan

1 Python Iterators

An iterator is an object that contains a countable number of values. It is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consists of the methods iter() and next().
Every time you ask an iterator for the **next** item, it calls its `__next__`method. If there is another value available, the iterator returns it. If not, it raises a `StopIteration` exception. More information about iterators can be found here.

This behavior (only returning the next element when asked to) has two main advantages:

1. Iterators need less space in memory. They remember the last value and a rule to get to the next value instead of memorizing every single element of a (potentially very long) sequence.
2. Iterators don't check how long the sequence they produce might get. For instance, they don't need to know how many lines a file has or how many files are in a folder to iterate through them.

(One important note: don't confuse iterators with iterables. Iterables are objects that can create iterators by using their `__iter__`method)

1. Building Custom Iterators

`__iter__()` and the `__next__()`

Building an iterator from scratch is easy in Python. We just have to implement the methods.

`__iter__()`

The method returns the iterator object itself. If required, some initialization can be performed.

`__next__()`

The method must return the next item in the sequence.

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

To iterate the characters of a string:

```
mystr = "banana"
 for x in mystr:
  print(x)
```

The `for` loop actually creates an iterator object and executes the next() method for each loop.

Classes and Objects

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

Defining a Class in Python

Like function definitions begin with the def keyword in Python, class definitions begin with a class keyword.
The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

Here is a simple class definition,

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')

# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

Creating Objects

The procedure to create an object is similar to a function call.

```
harry = Person()
```

Constructors in Python

Class functions that begin with double underscore __are called special functions as they have special meaning.
Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated.
This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')


# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)
```

We can even delete the object itself, using the del statement.

```
del c1
```

Inheritance in Python

Inheritance is a powerful feature in object oriented programming.
It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

Python Inheritance Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Python Stacks

A stack is a data structure that stores items in a Last-In/First-Out manner. We will look at three different implementations of stacks in Python.

- Using list data structure
- Using collections.deque module
- Using queue.LifoQueue class

Stacks via Lists

The built-in Python list object can be used as a stack. For stack **`.push()`** method, we can use **`.append()`** method of list. **`.pop()`** method of list can remove elements in LIFO order. Popping an empty list (stack) will raise an **`IndexError`.** To get the top most item (peek) in the stack, write **`list[-1]`**. Bigger lists (stacks) often run into speed issues as they continue to grow. **`list`** may be familiar, but it should be avoided because it can potentially have memory reallocation issues.

```
myStack = []

myStack.append('a')
myStack.append('b')
myStack.append('c')

myStack    # ['a', 'b', 'c']

myStack.pop()          # ‘c’
myStack.pop()          # ‘b’
myStack.pop()          # ‘a’
```

Stacks via collections.deque

This method solves the speed problem we face in lists. The **`deque`** class has been designed as such to provide O(1) time complexity for append and pop operations. The **`deque`** class is built on top of a doubly linked list structure which provides faster insertion and removal. Popping an empty **`deque`** gives the same **`IndexError`**. Read more about it here. Also, to know more about linked lists in Python, read this.

```
from collections import deque
myStack = deque()

myStack.append('a')
myStack.append('b')
myStack.append('c')
myStack    # deque(['a', 'b', 'c'])
myStack.pop()          # ‘c’
myStack.pop()          # ‘b’
myStack.pop()          # ‘a’
```

Stacks via queue.LifoQueue

`LifoQueue` uses **`.put()`** and **`.get()`** to add and remove data from the stack. **`LifoQueue`** is designed to be fully thread-safe. But use it only if you are working with threads. Otherwise, **`deque`** works well. The **`.get()`** method by default will wait until an item is available. That means it waits forever if no item is present in the list. Instead, **`get_nowait()`** method would immediately raise empty stack error. Read more about it here.

```
from queue import LifoQueue
myStack = LifoQueue()
```

```
myStack.put('a')
myStack.put('b')
myStack.put('c')

myStack    # <queue.LifoQueue object at 0x7f408885e2b0>

myStack.get()         # ‘c’
myStack.get()         # ‘b’
myStack.get()         # ‘a’
```

Python Queues

A queue is FIFO data structure. The insert and delete operations are sometimes called **`enqueue`** and **`dequeue`**. We can use list as a queue as well. To follow FIFO, use **`pop(0)`** to remove the first element of the queue. But as discussed before, lists are slow. They are not ideal from performance perspective.

We can use the **`collections.deque`** class again to implement Python queues. They work best for non-threaded programs. We can also use **`queue.Queue`** class. But it works well with synchronized programs.

If you are not looking for parallel processing, **`collections.deque`** is a good default choice.

```
from collections import deque
q = deque()
q.append('eat')
q.append('sleep')
q.append('code')
q          # deque(['eat', 'sleep', 'code'])
q.popleft()     # 'eat'
q.popleft()     # 'sleep'
q.popleft()     # 'code'
q.popleft()     # IndexError: "pop from an empty deque"
```

Exercise (30 marks)

1. For a list of integers, find square and cube for each value using lambda function.(20 marks)

2. Create a class for rectangle shape that calculates its area based upon the length and width (10 marks)

3. Form a queue such that it works in LIFO order (10 marks)