National University of Computer and Emerging Sciences

Lab Manual 04
CL461-Artificial Intelligence Lab

Course Instructor	Mr. Saif Ul Islam
Lab Instructor (s)	Hassan Masood Muhammad Adeel
Section	E
Semester	Spring 2023

Department of Computer Science
FAST-NU, Lahore, Pakistan

Table of Contents

1 Objectives

After performing this lab, students shall be able to understand Python data structures which include:

- Python lambda function, generators and decorators
- Python classes (inheritance) and dunder methods
- Python stacks & queues
- Python graphs

2 Task Distribution

Total Time	**170 Minutes**
Python Lambda Function, Decorators, Generators	30 Minutes
Python Classes & Dunder Methods	20 Minutes
Python Stacks & Queues	20 Minutes
Python Graphs	10 Minutes
Exercise	80 Minutes
Online Submission	10 Minutes

3 Python Lambda Functions

With Python, you can write functions on the go. Normally to define a function, you need `def` keyword, function header and body. But lambda functions offer a quicker way to write functions using fewer lines of code.

Also known as anonymous functions as they are declared without any name, the lambda function takes its name from the keyword `lambda` which is required to declare it. They behave in a similar way as a regular function behaves.

Some characteristics of lambda functions are:

- They are syntactically restricted to a single expression.
- A lambda function can take any number of arguments.
- A lambda expression can return a function.
- A lambda function can be passed as an argument within another higher-order function.

3.1 Structure

After the keyword **`lambda`**, we specify the names of the arguments; then, we use a colon followed by the expression that specifies what we wish the function to return.

```
lambda argument(s): expression
```

Example:

The following lambda function adds two numbers x and y.

```
lambda x, y: x+y
```

3.2 Lambda Function Usage

So how do we call an anonymous function? See below:

```
# Calling an anonymous function with a single argument
remainder = lambda num: num % 2
print(remainder(5))   #1

# Multiple parameters
product = lambda x, y : x * y
print(product(2, 3))

# As an argument to another function
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
pairs # [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Lambda functions are most commonly used in combination with **`map()`** and **`filter()`** functions.

i filter() usage

`filter()` method filters the given iterable with the help of a function object that tests each element in the iterable to be true or not. For its function argument, we normally use a lambda function.

Structure:

```
filter(function, iterable)
```

Example:

```
# Filter all list items greater than 7
numbers_list = [2, 6, 8, 10, 11, 4, 12, 7, 13, 17, 0, 3, 21]
filtered_list = list(filter(lambda num: (num > 7), numbers_list))
print(filtered_list)  # [8, 10, 11, 12, 13, 17, 21]
```

ii map() usage

Mapping consists of applying a transformation function to an iterable to produce a new iterable. Items in the new iterable are produced by calling the transformation function on each item in the original iterable. **`map()`** can be used as an alternate to loops.

Structure:

```
map(function, iterable[, iterable1, iterable2,..., iterableN])
```

Examples:

```
# Convert all the items in a list from a string to an integer number
str_nums = ["4", "8", "6", "5", "3", "2", "8", "9", "2", "5"]
int_nums = map(int, str_nums)
list(int_nums)  # [4, 8, 6, 5, 3, 2, 8, 9, 2, 5]

# Len of every string item in a list
words = ["Welcome", "to", "Real", "Python"]
list(map(len, words)) # [7, 2, 4, 6]

# Add multiple list
list(map(lambda x, y, z: x + y + z, [2, 4], [1, 3], [7, 8]))
# [10, 15]
```

4 Python Generators

Generators are used to create iterators, but with a different approach. Generator is a function that produces a sequence of results. It works by maintaining its local state using the **`yield`** keyword, so that the function can resume again exactly where it left off when called subsequent times. Thus, you can think of a generator as something like a powerful iterator. They offer simplified code and on-demand calculations. This means your program can use only the values needed without having to wait until all of them have been generated. Explore generator expressions yourself.

Examples:

```
def myGenerator1(n):
    for i in range(n):
        yield i

def myGenerator2(n, m):
    for j in range(n, m):
        yield j
```

```
def myGenerator3(n, m):
    yield from myGenerator1(n)
    yield from myGenerator2(n, m)
    yield from myGenerator2(m, m+5)

print(list(myGenerator1(5)))     # [0, 1, 2, 3, 4]
print(list(myGenerator2(5, 10))) # [5, 6, 7, 8, 9]
print(list(myGenerator3(0, 10)))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

4.1 Difference between Return and Yield

The keyword **`return`** returns a value from a function, at which time the function then loses its local state. Thus, the next time we call that function, it starts over from its first statement.

On the other hand, **`yield`** maintains the state between function calls, and resumes from where it left off when we call the **`next()`** method again. So if **`yield`** is called in the generator, then the next time the same generator is called we'll pick right back up after the last **`yield`** statement.

5 Python Decorators

A decorator is a design pattern in Python that allows a user to dynamically add new functionality to an existing function object without modifying its structure. Functions support operations such as being passed as an argument, returned from a function, modified, and assigned to a variable. This provides the basis of decorators. Decorators make an extensive use of closures.

Examples:

```
# Decorators with arguments
def decorator_with_arguments(function):
    def wrapper_accepting_arguments(arg1, arg2):
        print("My arguments are: {0}, {1}".format(arg1,arg2))
        function(arg1, arg2)
    return wrapper_accepting_arguments

@decorator_with_arguments
def cities(city_one, city_two):
    print("Cities I love are {0} and {1}".format(city_one, city_two))

cities("Nairobi", "Accra")
# My arguments are: Nairobi, Accra
# Cities I love are Nairobi and Accra

# Multiple decorators
```

```
def split_string(function):
    def wrapper():
        func = function()
        splitted_string = func.split()
        return splitted_string

    return wrapper
# Notice the order in which the decorators are applied
@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'

say_hi()   # ['HELLO', 'THERE']
```

6 Python Classes

Python is an "object-oriented programming language." Not purely. Like C++. But much of it works in a similar manner to what you have learned previously with other languages. It is flexible enough and powerful enough to allow you to build your applications using the object-oriented paradigm.

The keyword **`class`** is used to define a class. When you define methods, you will need to always provide the first argument to the method with a **`self`** keyword. This keyword accesses the class attributes. "Private" instance variables that cannot be accessed except from inside an object don't exist in Python.

Examples:

```
# Simple class
class myClass():
  str = "Cool"
  def method1(self):
      print("Artificial Intelligence")

  def method2(self,someString):
      self.str = someString
      print("AI Lab:" + someString)


def main():
  c = myClass()
  c.method1()
  c.method2("AI Lab is fun")
```

```
if __name__== "__main__":
  main()
```

You can also provide the values for the attributes at runtime. This is done by defining the attributes inside the **`__init__`** method as knows as the class constructor. The following example illustrates this.

```
# Class with __init__ method
class Snake:
    def __init__(self, name):
        self.name = name

    def change_name(self, new_name):
        self.name = new_name

# two variables are instantiated
python = Snake("python")
anaconda = Snake("anaconda")

# print the names of the two variables
print(python.name)    # python
print(anaconda.name)  # anaconda
```

6.1 Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. Derived classes may override methods of their base classes. All methods in Python are effectively virtual. **`super()`** is used to initialize the members of the base class.

Example:

```
class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyEmployee(Employee):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)
        self.hours_worked = hours_worked
```

```
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

Learn more about Python inheritance and composition here. Also, checkout multiple inheritance here.

7 Python Dunder

"Dunder", also known as "Special Methods" or "Magic Methods", is the common pronunciation for python's built-in method names that start and end with double underscores. Since "Dunder" is easier to say than "double underscore", the name stuck. The only thing magic or special about these methods is that they allow you to include built-in type behavior in your custom classes. Few dunder methods are explained below:

- **`__new__`** is called first to create a new instance of your class.
- **`__init__`** method is called to initialize that newly created instance.
- **`__str__`** method will return an "informal" printable representation of an object and return str type. The **`__str__`** dunder method is called by **`str()`** as well as the built-in format() and print() methods. Meaning, that anytime you use **`print()`** you're also calling the **`__str__`** method on whatever object you're trying to print.
- **`__repr__`** is used to return the "official" string representation of an object.
- **`__eq__`** is the dunder method used for checking equality between objects.

Examples:

```
class WidgitWithoutStr:
    """
    A class with no __str__ or __repr__ methods defined.
    """
    def __init__(self, name):
        self.name = name

class WidgitWithStrOnly(WidgitWithoutStr):
    """
```

```
    A class with __str__ defined.
    """
    def __str__(self):
        return self.name

class WidgitWithReprOnly(WidgitWithoutStr):
    """
    A class with __repr__ defined.
    """
    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.name)

print(WidgitWithoutStr("Nobody")) # <__main__.WidgitWithoutStr object
at 0x10b8c1790>
print(WidgitWithStrOnly("Bob")) # Bob
print(WidgitWithReprOnly("Mary")) # WidgitWithReprOnly(Mary)
```

Learn more about dunders here. To understand the role of underscore(_) in Python, visit this link.

8 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

8.1 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’
```

8.2 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’
```

8.3 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’
```

9 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"
```

10 Python Graphs

Graphs are networks consisting of nodes connected by edges or arcs. In Python, we can represent graphs using dictionary data structure. The keys become nodes and their corresponding values become edges. We can use dictionary syntax to make our custom graph data structure along with all of its methods. Graphs are excellent for implementing path-finding algorithms.

There are a few libraries which provide graph data structure using Python e.g. python-graph. For self-exploration.

Graph Structure:

```
graph = {'A': ['B', 'C'],
     'B': ['C', 'D'],
     'C': ['D'],
     'D': ['C'],
     'E': ['F'],
     'F': ['C']}
```

Simple Graph Code:

```
class graph:
  def __init__(self,gdict=None):
    if gdict is None:
      gdict = {}
```

```
    self.gdict = gdict

  def getVertices(self):
    return list(self.gdict.keys())

  # Add the vertex as a key
  def addVertex(self, vrtx):
    if vrtx not in self.gdict:
      self.gdict[vrtx] = []

  def edges(self):
    return self.findedges()

  # Add the new edge
  def AddEdge(self, edge):
    edge = set(edge)
    (vrtx1, vrtx2) = tuple(edge)
    if vrtx1 in self.gdict:
      self.gdict[vrtx1].append(vrtx2)
    else:
      self.gdict[vrtx1] = [vrtx2]

  # List the edge names
  def findedges(self):
    edgename = []
    for vrtx in self.gdict:
      for nxtvrtx in self.gdict[vrtx]:
        if {nxtvrtx, vrtx} not in edgename:
            edgename.append({vrtx, nxtvrtx})
    return edgename
```

11 Exercise (30 marks)

11.1 For a list of integers, find square and cube for each value using lambda function (10 marks)

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

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

After creation of the class, define the relevant attributes. Define a function for area computation and then a function for displaying area. Incorporate your knowledge of class and objects here.

12 Submission Instructions

Always read the submission instructions carefully.

- Rename your Jupyter notebook to your roll number and download the notebook as **.ipynb** extension.
- To download the required file, go to **File->Download .ipynb**
- Only submit the **.ipynb** file. DO NOT **zip** or **rar** your submission file.
- Submit this file on Google Classroom under the relevant assignment.
- Late submissions will not be accepted.