National University of Computer and Emerging Sciences

Artificial Intelligence

Lab 4

Fast School of Computing

FAST-NU, Lahore, Pakistan

Objectives

- A review on Python Iterators, Python Classes & Objects, Constructors, & Inheritance
- A review on Python stacks & queues
- A* Search
- Hill Climbing Search
- Exercises

Contents

1. Python Iterators

An iterator in Python is an object designed to hold a countable number of values. It enables iteration, allowing traversal through all the stored values. Technically, an iterator in Python adheres to the iterator protocol, which consists of the methods `iter()` and `next()`.

When you request the next item from an iterator, it invokes its `__next__()` method. If there's another value available, the iterator returns it; otherwise, it raises a `StopIteration` exception.

This approach offers two primary advantages:

1. Iterators consume less memory as they retain the last value and a rule to derive the subsequent value, instead of storing every element of a potentially extensive sequence.
2. Iterators do not need to determine the length of the sequence they produce. For example, they do not require knowledge of the number of lines in a file or the number of files in a folder to iterate through them.

It's essential not to confuse iterators with iterables. Iterables are objects capable of generating iterators by employing their `__iter__()` method.

1.1 Building Custom Iterators

In Python, building an iterator from scratch involves implementing two methods:

- `__iter__()`: This method returns the iterator object itself, and optionally, any necessary initialization can be performed within it.
- `__next__()`: This method is responsible for returning the next item in the sequence.

Here's how you can create an iterator and iterate through its values:

```
# Define a tuple
mytuple = ("apple", "banana", "cherry")

# Create an iterator from the tuple
myit = iter(mytuple)

# Print each value of the iterator
print(next(myit))
print(next(myit))
print(next(myit))
```

output:

```
apple
banana
cherry
```

You can also iterate through the characters of a string using a for loop:

```
# Define a string
mystr = "banana"

# Iterate through the characters of the string
for x in mystr:
    print(x)
```

output:

```
b
```

```
a
n
a
n
a
```

This demonstrates how Python handles iteration using iterators, and how you can create and use them effectively.

2. Python Classes & 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.

2.1 Defining a Class in Python

In Python, class definitions start with the `class` keyword, similar to how function definitions begin with the `def` keyword. The first string inside a class, known as the docstring, provides a brief description of the class. While not mandatory, it's highly recommended to include one. Here's an example of a simple class definition:

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

When a class is defined, it creates a new local namespace where all its attributes, whether data or functions, are defined.
Upon defining a class, a new class object is created with the same name. This class object enables access to various attributes and allows instantiation of new objects of that class.
For instance:

```
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__)
```

In this example, accessing **Person.age** returns **10**, accessing **Person.greet** returns **<function Person.greet>**, and accessing **Person.__doc__** returns **"This is a person class"**. These illustrate the attributes and methods associated with the **Person** class.

2.2 Creating Objects

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

```
harry = Person()
```

3. Constructors in Python

In Python, class functions that begin with double underscores (`__`) are known as special functions because they hold special significance in the context of object-oriented programming (OOP). One particularly important special function is `__init__()`, which serves as the constructor method. This method is automatically invoked whenever a new object of that class is instantiated. Typically, `__init__()` is used to initialize the attributes of the class.

Here's an example illustrating the use of `__init__()` in a class called `ComplexNumber`:

```
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)
```

In this example, `num1` is an instance of the `ComplexNumber` class with the real part `2` and the imaginary part `3`.

Additionally, you can delete objects in Python using the `del` statement. For instance, to delete the object `num1`, you would use:

```
del num1
```

This would remove the reference to the `num1` object, allowing Python's garbage collector to reclaim the memory allocated for that object.

4. 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.

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

5. Python Stacks

In Python, a stack is a data structure that follows the Last-In/First-Out (LIFO) principle, where the last element added to the stack is the first one to be removed. There are several ways to implement a stack in Python:

5.1 Using a list data structure:

You can implement a stack using Python's built-in list data structure. You can use the `append()` method to push elements onto the stack and the `pop()` method to remove elements from the top of the stack.

```
stack = []

# Push items onto the stack
stack.append(1)
stack.append(2)
stack.append(3)

# Pop items from the stack
print(stack.pop())  # Output: 3
print(stack.pop())  # Output: 2
print(stack.pop())  # Output: 1
```

5.2 Using the collections.deque module:

Python's `collections.deque` class provides an optimized implementation of a double-ended queue. You can use it to implement a stack by using its `append()` and `pop()` methods.

```
from collections import deque

stack = deque()

# Push items onto the stack
stack.append(1)
stack.append(2)
stack.append(3)

# Pop items from the stack
print(stack.pop())  # Output: 3
print(stack.pop())  # Output: 2
print(stack.pop())  # Output: 1
```

5.3 Using the queue.LifoQueue class:

The `queue` module provides the `LifoQueue` class, which is specifically designed for implementing a stack. You can use its `put()` method to push items onto the stack and the `get()` method to pop items from the stack.

```
from queue import LifoQueue

stack = LifoQueue()
```

```
# Push items onto the stack
stack.put(1)
stack.put(2)
stack.put(3)

# Pop items from the stack
print(stack.get())  # Output: 3
print(stack.get())  # Output: 2
print(stack.get())  # Output: 1
```

These are three different implementations of a stack in Python, each with its advantages and use cases. You can choose the implementation that best suits your requirements based on factors such as performance and functionality.

6. Python Queues

A queue follows the First-In, First-Out (FIFO) principle, where items are inserted at the back and removed from the front. Common operations include enqueue for insertion and dequeue for removal. While lists can serve as queues using `pop(0)` for FIFO behavior, they may not offer optimal performance due to their implementation.

Python provides alternative methods for implementing queues, such as using the `collections.deque` class, which offers efficient FIFO operations and is suitable for non-threaded programs. Additionally, the `queue.Queue` class can be utilized, particularly in synchronized programs or those involving parallel processing.

Here's an example demonstrating the use of `collections.deque` for implementing a queue:

```
from collections import deque
q = deque()
q.append('eat')
q.append('sleep')
q.append('code')

# Output: deque(['eat', 'sleep', 'code'])
print(q)

# Output: 'eat'
print(q.popleft())

# Output: 'sleep'
print(q.popleft())

# Output: 'code'
print(q.popleft())

# Output: IndexError: "pop from an empty deque"
print(q.popleft())
```

In this example, elements are added to the queue using `append()`, and elements are removed from the left side of the queue using `popleft()`. Attempting to remove an element from an empty queue raises an `IndexError`. This demonstrates the FIFO behavior of the queue implemented using `collections.deque`.

7. Optimizing Mathematical Functions using Hill Climbing Search

You are tasked with optimizing a mathematical function using the Hill Climbing Search (HCS) algorithm. Given a continuous mathematical function ($f(x)$), your objective is to find the value of (x) that maximizes or minimizes ($f(x)$) within a specified range.

Task:

Implement the Hill Climbing Search algorithm to find the optimal value of (x) for the given mathematical function ($f(x)$). The algorithm should iteratively adjust the value of (x) to move towards the optimal solution, climbing the slope of the function until it reaches a peak (for maximization) or a valley (for minimization).

Constraints:

- The function ($f(x)$) should be continuous within the specified range.
- The search space for (x) should be defined within a specific interval.

8. Exercises

8.1 Rectangle Class (10 marks)

Create a class that represents a rectangle shape and implements methods to calculate its area, perimeter, diagonal length, aspect ratio based on the given length and width.

8.2 Adding A* method in Cube Solver Class (40 marks)

*Repeat Cube Finder Question from previous labs using **A*** search.*

In this cube puzzle, represented by a 2D array with 0s for open spaces, 1s for high walls, and 2s for short walls, the goal is to employ an A* search algorithm to determine the shortest path from the starting point (S) to the goal point (G). The task emphasizes selecting paths with the least number of encounters with short walls or no walls. Movement is confined to four directions (up, down, left, right), and the puzzle requires navigating around high walls while allowing jumps over short walls. The objective is to implement the A* search algorithm, read the cube puzzle from a text file, and provide the shortest path from the starting point to the goal, prioritizing routes with minimal short wall encounters.

8.3 Optimizing a function using HCS (25+25 = 50 marks)

Consider the mathematical function ($f(x) = x^2 - 4x + 5$) defined within the range ($0 \leq x \leq 5$).

Task 1: Maximization

Your task is to use the Hill Climbing Search algorithm to find the value of (x) that maximizes ($f(x)$) within the range \($0 \leq x \leq 5$).

Task 2: Minimization

Your task is to use the Hill Climbing Search algorithm to find the value of (x) that minimizes ($f(x)$) within the same range ($0 \leq x \leq 5$).

Output:

For both tasks, provide the optimal value of (x) that maximizes or minimizes $(f(x))$, along with the corresponding maximum or minimum value of $(f(x))$ achieved.

Note:

Ensure to properly initialize the starting point for the hill climbing algorithm within the defined range. Implement mechanisms to handle different types of functions and ranges efficiently within the Hill Climbing Search algorithm to ensure robust optimization results.

References

[1] AI Search Algorithms With Examples | by Pawara Siriwardhane, UG | Nerd For Tech | Medium
[2] Uninformed Search: BFS, DFS, DLS and IDS (substack.com)
[3] https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
[4] https://thepythoncodingbook.com/2021/10/31/using-lists-tuples-dictionaries-and-sets-in-python/