National University of Computer and Emerging Sciences

Artificial Intelligence

Lab 3

Fast School of Computing

FAST-NU, Lahore, Pakistan

Objectives

- A review on Python Sets, Exception Handling & File Handling
- A review on DLS, IDS & UCS
- Exercises

Contents

1. Python Sets

Sets have following characteristics:

- Set in Python is a data structure equivalent to sets in mathematics.
- Sets are a mutable collection of distinct (unique) immutable values that are unordered.
- Any immutable data type can be an element of a set: a number, a string, a tuple.
- Mutable (changeable) data types cannot be elements of the set.
- In particular, list cannot be an element of a set (but tuple can), and another set cannot be an element of a set.
- You can perform standard operations on sets (union, intersection, difference).

1.1 Set Initialization

You can initialize a set in the following ways:

```
# Initialize empty set
emptySet = set()
# Pass a list to set() to initialize it
dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])
dataEngineer  = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'])
# Direct initialization using curly braces
dataScientist = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'}
dataEngineer  = {'Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'}
# Curly braces can only be used to initialize a set containing values
emptyDict= {}
```

1.2 Set Modification

Let's consider the following set for our add/remove examples:

```
# Initialize set with values
graphicDesigner = {'InDesign', 'Photoshop', 'Acrobat', 'Premiere', 'Bridge'}
# Add a new immutable element to the set
graphicDesigner.add('Illustrator')
# TypeError: unhasable type 'list'
graphicDesigner.add(['Powerpoint', 'Blender'])
# Remove an element from the set
graphicDesigner.remove('Illustrator')
# Another way to remove an element. What is the difference?
graphicDesigner.discard('Premiere')
# Remove and return an arbitrary value from a set
graphicDesigner.pop()
# Remove al values from the set
graphicDesigner.clear()
```

1.3 Set Operations

Python sets have methods that allow you to perform these mathematical operations like union, intersection, difference, and symmetric difference.

Let's initialize two sets to work on our examples:

```
# Initialize sets
dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])
dataEngineer = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'])
# set built-in function union
```

```
dataScientist.union(dataEngineer)
# Equivalent Result
dataScientist | dataEngineer
# Intersection operation
dataScientist.intersection(dataEngineer)
# Equivalent Result
dataScientist & dataEngineer
# These sets have elements in common so isdisjoint would return False
dataScientist.isdisjoint(dataEngineer)
# Difference Operation
dataScientist.difference(dataEngineer)
# Equivalent Result
dataScientist - dataEngineer
# Symmetric Difference Operation
dataScientist.symmetric_difference(dataEngineer)
# Equivalent Result
dataScientist ^ dataEngineer
```

1.4 Frozensets

You have encountered nested lists and tuples. The problem with nested sets is that you cannot normally have nested sets as sets cannot contain mutable values including sets.

- A frozenset is very similar to a set except that a frozenset is immutable.
- The primary reason to use them is to write clearer, functional code.
- By defining a variable as a frozen set, you're telling future readers: do not modify this.
- If you want to use a frozen set you'll have to use the function to construct it. No other way.

```
# Nested Lists and Tuples
nestedLists = [['the', 12], ['to', 11], ['of', 9], ['and', 7], ['that', 6]]
nestedTuples = (('the', 12), ('to', 11), ('of', 9), ('and', 7), ('that', 6))
# Initialize a frozenset
immutableSet = frozenset()
# Initialize a frozenset
nestedSets = set([frozenset()])
```

A major disadvantage of a frozenset is that since they are immutable, it means that you cannot add or remove values.

```
# AttributeError: 'frozenset' object has no attribute 'add'
immutableSet.add("Strasbourg")
```

2. Python Exception Handling

An exception is an error that is thrown by our code when the execution of the code results in an unexpected outcome. Normally, an exception will have an error type and an error message. Some examples are as follows.

```
ZeroDivisionError: division by zero
TypeError: must be str, not int
```

ZeroDivisionError and TypeError are the error type and the text that comes after the colon is the error message. The error message usually describes the error type.

2.1 Types of Exceptions

Here's a list of the common exceptions you'll come across in Python:

1. **ImportError**: It is raised when you try to import the library that is not installed or you have provided the wrong name
2. **IndexError**: Raised when an index is not found in a sequence. For example, if the length of the list is 10 and you are trying to access the 11th index from that list, then you will get this error
3. **IndentationError**: Raised when indentation is not specified properly
4. **ZeroDivisionError**: It is raised when you try to divide a number by zero
5. **ValueError**: Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified
6. **Exception**: Base class for all exceptions. If you are not sure about which exception may occur, you can use the base class. It will handle all of them.

2.2 Exception Handling with Try Except Clause

Python provides us with the try except clause to handle exceptions that might be raised by our code. The basic anatomy of the try except clause is as follows:

```
try:
    // some code
except:
    // what to do when the code in try raise an exception
```

In plain English, the try except clause is basically saying, "Try to do this, except (otherwise) if there's an error, then do this instead".

There are a few options on what to do with the thrown exception from the try block. Let's discuss them.

2.3 Re-raise the exception

Let's take a look at how to write the try except statement to handle an exception by re-raising it. First, let's define a function that takes two input arguments and returns their sum.

```
def myfunction(a, b):
    return a + b
```

Next, let's wrap it in a try except clause and pass input arguments with the wrong type so the function will raise the TypeError exception.

```
try:
    myfunction(100, "one hundred")
except:
    raise Traceback (most recent call last)
TypeError: unsupported operand type(s) for +: 'int' and 'str'

During handling of the above exception, another exception occurred:

NameError                                 Traceback (most recent call last)
Cell In[23], line 6
      4     myfunction(100, "one hundred")
      5 except:
----> 6     raise Traceback("most recent call last")

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

2.4 Catch certain types of exception

Another option is to define which exception types we want to catch specifically. To do this, we need to add the exception type to the except block.

```
try:
    myfunction(100, "one hundred")
except TypeError:
    print("Cannot sum the variables. Please pass numbers only.")

Cannot sum the variables. Please pass numbers only.
```

To make it even better, we can actually log or print the exception itself.

```
try:
    myfunction(100, "one hundred")
except TypeError as e:
    print(f"Cannot sum the variables. The exception was:{e}")

Cannot sum the variables. The exception was:unsupported operand type(s) for
+: 'int' and 'str'
```

Furthermore, we can catch multiple exception types in one except clause if we want to handle those exception types the same way. Let's pass an undefined variable to our function so that it will raise the NameError. We will also modify our except block to catch both TypeError and NameError and process either exception type the same way.

```
try:
    myfunction(100, a)
except (TypeError, NameError) as e:
    print(f"Cannot sum the variables. The exception was {e}")

Cannot sum the variables. The exception was name 'a' is not defined
```

2.5 Try…. Finally

So far, the try statement had always been paired with except clauses. But there is another way to use it as well. The try statement can be followed by a finally clause. Finally, clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "finally" clause is always executed regardless if an exception occurred in a try block or not. A simple example to demonstrate the finally clause:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)

Your number:  34
There may or may not have been an exception.
The inverse:  0.029411764705882353
```

2.6 Try..except and finally

"finally" and "except" can be used together for the same try block, as it can be seen in the following Python example:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

```
Your number:  23
There may or may not have been an exception.
```

3. Python File Handling

Python allows users to handle files by supporting to read and write files, along with many other file handling options.

3.1 Open & Close a file

When you want to read or write a file, the first thing to do is to open the file. Python has a built-in function open that opens the file and returns a file object. To return a file object we use open() function along with two arguments, that accepts file name and the mode, whether to read or write.

The syntax is given below:

```
open(filename, mode)
```

3.2 Kinds of modes

There are three basic types of modes in which files can be opened in Python.

mode	Description
r	open for reading (default)
r+	open for both reading and writing (file pointer is at the beginning of the file)
w	open for writing (truncate the file if it exists)
w+	open for both reading and writing (truncate the file if it exists)
a	open for writing (append to the end of the file if exists & file pointer is at the end of the file)

Always keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be " r " by default.

Let's look at this program and try to analyze how the read mode works:

```
# a file named "book", will be opened with the reading mode.
file = open('book.txt', 'r')
# This will print every line one by one in the file
for each in file:
    print(each)
```

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
```

```
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat.
```

3.3 Working of read() mode

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use file.read(). The full code would work like this:

```
# Python code to illustrate read() mode
file = open("book.txt ", "r")
print (file.read())
```

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat.
```

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("book.txt", "r")
print (file.read(5))
```

```
Lorem
```

3.4 Working of write() mode

Let's see how to create a file and how write mode works:

To manipulate the file, write the following in your Python environment:

```
# Python code to create a file
file = open('book.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

The close() command terminates all the resources in use and frees the system of this particular program.

3.5 Working of write() mode

```
# Python code to illustrate append() mode
file = open('book.txt','a')
file.write("This will add this line")
file.close()
```

4. DLS, IDS & UCS

Continue considering staircase example from AI_Lab2

DFS is not always bad, in fact, with the right improvement and problem, it can outperform a BFS. Yes, you at the back, "What kind of improvements can be made to it?", you asked. Well, I'm glad you asked. We know that the major setback of DFS is the fact that it can get caught up doing unnecessary work on staircase 1 when the solution we seek is on staircase 2, so can we tell it when to stop looking on staircase 1 and start looking at staircase 2? Yes. Yes, we can (or whatever Obama said).

Depth Limited Search (DLS) is an improvement on DFS that allows us to state the maximum depth (number of steps in our case) that we should go to on staircase 1 before moving to staircase 2.

If we used DLS instead of DFS on our staircase problem, we could say the maximum depth is 4 so our path to k would then be a→b→e→g →c→d→j→k. That is, we go down staircase 1 till we get to step 4,

then we stop and start looking at staircase 2. As shown, this provides a shorter path to k when compared to the DFS path.

Iterative Deepening Search (IDS) is Depth Limited Search on steroids. Simply put, IDS is DLS in a loop. Instead of providing a static maximum depth as we did in depth limited search, we loop from 1 to the expected maximum provided maximum depth.

While IDS does in fact conduct repetitive work, it can provide better results. If the maximum depth provided for IDS is 4, we will eventually end up with the same solution as the DLS provided above, but that is only because letter k is on the 4th step.

For better clarity, let's assume we are looking for the letter c.

Provided that the maximum depth is 4, DLS will result in our path being a→b→e→g →c because we will go down 4 steps on staircase 1 before starting on staircase 2. However, IDS will give us a→c because we iterate from 1-4, each time providing the current step as the maximum depth.

Uniform Cost Search (UCS) is a graph search algorithm that explores nodes in a weighted graph by expanding the node with the lowest path cost. It systematically examines all possible paths to find the optimal solution with the least total cost.

5. Exercises

5.1 Set Operations (10 marks)

Consider two sets X and Y. You may take any type of values for these sets. Try to find a solution to get a set having all elements in either X or Y, but not both.

5.2 Exception Handling for Division (10 marks)

Write a function to divide two numbers P and Q. Implement exception handling technique (try..except clause) for handling possible exceptions in the scenario.

5.3 Reading text from a file and storing it in reversed order (10 marks)

Design a code which reads text from the file "Alphabets.txt" and stores its data in reverse order in another file. For this you may upload the given text file on Google Collab's session and define the path as:

```
file_path= '/Alphabets.txt'
```

The same convention can be followed for defining path of the resultant file (reversed text file).

5.4 Cube Finder (20 marks)

Repeat Cube Finder Question from AI_Lab2 with IDS.

Note: Implement the IDS algorithm by repeatedly performing DLS with increasing depth limits until the goal state is found. DLS should accept the cube, depth limit, and current depth as inputs.

5.5 Cube Finder (20 marks)

Repeat Cube Finder Question from AI_Lab2 with UCS.

Note: Implement the Uniform Cost Search algorithm to explore the lowest cost nodes first. Use a priority queue to keep track of the nodes to explore.

5.6 CubeFinder Class (30 marks)

Design a CubeFinder class to solve the cube puzzle problem using various search algorithms. The CubeFinder class should contain definitions for the following functions:

1. __init__(self, filename: str): Initializes the CubeFinder object with the filename of the text file containing the cube representation.
2. read_cube_from_file(self): Reads the cube from the text file specified during initialization and returns a 2D list representing the cube.
3. dfs_search(self): Implements DFS to find a path from the starting point to the goal point in the cube. Returns the path as a list of coordinates or -1 if no path exists.
4. bfs_search(self): Implements BFS to find a path from the starting point to the goal point in the cube. Returns the path as a list of coordinates or -1 if no path exists.
5. dls_search(self, depth_limit: int): Implements DLS with a specified depth limit to find a path from the starting point to the goal point in the cube. Returns the path as a list of coordinates or -1 if no path exists.
6. ids_search(self): Implements IDS to find a path from the starting point to the goal point in the cube. Returns the path as a list of coordinates or -1 if no path exists.
7. ucs_search(self): Implements UCS to find a path from the starting point to the goal point in the cube. Returns the path as a list of coordinates or -1 if no path exists.

The main function should create an instance of the CubeFinder class with the specified filename, and then call a specific search algorithm (DFS, BFS, DLS, IDS, or UCS) based on user input. Finally, it should print the path found by the selected search algorithm.
Design the CubeFinder class and the main function accordingly.

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/