National University of Computer and Emerging Sciences

Artificial Intelligence

Lab 2

Fast School of Computing

FAST-NU, Lahore, Pakistan

Objectives

- A review on Python Lists, Tuples, & Dictionaries
- A review on BFS, DFS
- Exercises

Contents

1. Python Lists

Everything in Python is treated as an object. Lists in Python represent ordered sequences of values. Lists are "mutable", meaning they can be modified "in place". You can access individual list elements with square brackets. Python uses zero-based indexing, so the first element has index 0.

Here are a few examples of how to create lists:

```
# List of integers
primes = [2, 3, 5, 7]

# We can put other types of things in lists
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn',
'Uranus', 'Neptune']
```

```
# We can even make a list of lists
hands = [
    ['J', 'Q', 'K'],
    ['2', '2', '2'],
    ['6', 'A', 'K'], # (Comma after the last element is optional)
]

# A list can contain a mix of different types of variables:
my_favourite_things = [32, 'AI Lab, 100.25]
```

1.1 Indexing & Slicing Examples

Consider our list of planets created above:

```
planets[0]     # 'Mercury'
planets[1]     # 'Venus'
planets[-1]    # 'Neptune'
planets[-2]    # 'Uranus'

# List Slicing

# first three planets
planets[0:3]   # ['Mercury', 'Venus', 'Earth']
planets[:3]    # ['Mercury', 'Venus', 'Earth']

# All the planets from index 3 onward
planets[3:]    # ['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

# All the planets except the first and last
planets[1:-1]  # ['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']

# The last 3 planets
planets[-3:]   # ['Saturn', 'Uranus', 'Neptune']
```

1.2 Indexing & Slicing Examples

Working with the same planets list:

```
# Rename Mars
planets[3] = 'Malacandra'
# ['Mercury', 'Venus', 'Earth', 'Malacandra', 'Jupiter', 'Saturn', 'Uranus',
'Neptune']

# Rename multiple list indexes
planets[:3] = ['Mur', 'Vee', 'Ur']
['Mur', 'Vee', 'Ur', 'Malacandra', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

1.3 List functions

Python has several useful functions for working with lists.

```
len(planets)      # 8

# The planets sorted in alphabetical order
```

```
sorted(planets)
# ['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus',
'Venus']

primes = [2, 3, 5, 7]
sum(primes) # 17
max(primes) # 7

# Let’s add Pluto to the planets list
planets.append('Pluto')

# Pop removes and returns the last element of the list
planets.pop()     # ‘Pluto’

# Remove an item from a list given its index instead of its value
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]    # [1, 66.25, 333, 333, 1234.5]

# Remove slices from the list
del a[2:4]  # [1, 66.25, 1234.5]

planets.index('Earth')  # 2

# Is Earth a planet?
"Earth" in planets      # True

# Is Pluto a planet?
"Pluto" in planets      # False (We removed it remember)

# Finally to find all the methods associated with Python list object
help(planets)
```

1.4 List functions

List comprehensions are one of Python's most unique features. List comprehensions combined with functions like min, max, and sum can lead to impressive one-line solutions for problems that would otherwise require several lines of code. The easiest way to understand them is probably to just look at a few examples:

```
# With list comprehension
squares = [n**2 for n in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Without list comprehension
squares = []
for n in range(10):
    squares.append(n**2)

# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
# List comprehensions are great of filtering and transformations
short_planets = [planet for planet in planets if len(planet) < 6]
# ['Venus', 'Earth', 'Mars']

[
    planet.upper() + '!'
    for planet in planets
    if len(planet) < 6
]
# ['VENUS!', 'EARTH!', 'MARS!']

# One line solution
def count_negatives(nums):
    # False + True + True + False + False equals to 2.
    # return len([num for num in nums if num < 0])
    return sum([num < 0 for num in nums])

count_negatives([5, -1, -2, 0, 3])
```

2. Python Tuples

Tuples are almost exactly the same as lists. They differ in just two ways.

1. The syntax for creating them uses parentheses instead of square brackets.
2. They cannot be modified (they are immutable).

Tuples are often used for functions that have multiple return values.

```
t = (1, 2, 3)
t = 1, 2, 3 # equivalent to above
t[0] = 100  # TypeError: 'tuple' object does not support item assignment

# Classic Python Swapping Trick
a = 1
b = 0
a, b = b, a # 0 1
```

2.1 Tuple Functions

There are only two tuple methods count() and index() that a tuple object can call.

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)  # 2

thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.index(8)  # 3
```

3. Python Dictionaries

Dictionaries and lists share the following characteristics:

- Both are mutable.
- Both are dynamic. They can grow and shrink as needed.

- Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

- List elements are accessed by their position in the list, via indexing.
- Dictionary elements are accessed via keys not by numerical index.

Duplicate keys are not allowed. A dictionary key must be of a type that is immutable. E.g. a key cannot be a list or a dict.

Here are a few examples to create dictionaries:

```
MLB_team = {
     'Colorado' : 'Rockies',
     'Boston'   : 'Red Sox',
     'Minnesota': 'Twins',
     'Milwaukee': 'Brewers',
     'Seattle'  : 'Mariners'
}
# Can also be defined as:
MLB_team = dict([
     ('Colorado', 'Rockies'),
     ('Boston', 'Red Sox'),
     ('Minnesota', 'Twins'),
     ('Milwaukee', 'Brewers'),
     ('Seattle', 'Mariners')
])
# Another way
tel = dict(sape=4139, guido=4127, jack=4098)

# dict comprehensions can be used to create dictionaries from arbitrary key
and value expression
{x: x**2 for x in (2, 4, 6)}  # {2: 4, 4: 16, 6: 36}

# Building a dictionary incrementally – if you don’t know all the key-value
pairs in advance
person = {}
person['fname'] = 'Joe'
person['lname'] = 'Fonebone'
person['age'] = 51
person['spouse'] = 'Edna'
person['children'] = ['Ralph', 'Betty', 'Joey']
person['pets'] = {'dog': 'Fido', 'cat': 'Sox'}
# {'fname': 'Joe', 'lname': 'Fonebone', 'age': 51, 'spouse': 'Edna',
'children': ['Ralph', 'Betty', 'Joey'], 'pets': {'dog': 'Fido', 'cat':
'Sox'}}
```

3.1 Dictionary Modification Examples

A few examples to access the dictionary elements, add new key value pairs, or update previous value:

```
# Retrieve a value
```

```
MLB_team['Minnesota']   # 'Twins'

# Add a new entry
MLB_team['Kansas City'] = 'Royals'

# Update an entry
MLB_team['Seattle'] = 'Seahawks'
```

3.2 Dictionary Formatting Examples

The % operator works conveniently to substitute values from a dict into a string by name:

```
hash = {}
hash['word'] = 'garfield'
hash['count'] = 42
s = 'I want %(count)d copies of %(word)s' % hash  # %d for int, %s for string
# 'I want 42 copies of garfield'
```

3.3 Dictionary Functions

The following is an overview of methods that apply to dictionaries:

```
# Let's use this dict for to demonstrate dictionary functions
d = {'a': 10, 'b': 20, 'c': 30}

# Clears a dictionary.
d.clear()   # {}

# Returns the value for a key if it exists in the dictionary.
print(d.get('b')) # 20

# Removes a key from a dictionary, if it is present, and returns its value.
d.pop('b')  # 20

# Returns a list of key-value pairs in a dictionary.
list(d.items())   # [('a', 10), ('b', 20), ('c', 30)]
list(d.items())[1][0]   # 'b'
list(d.items())[1][1]   # 20

# Returns a list of keys in a dictionary.
list(d.keys())    # ['a', 'b', 'c']

# Returns a list of values in a dictionary.
list(d.values())  # [10, 20, 30]

# Removes the last key-value pair from a dictionary.
d.popitem() # ('c', 30)

# Merges a dictionary with another dictionary or with an iterable of key-
value pairs.
d2 = {'b': 200, 'd': 400}
d.update(d2)      # {'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

For more details, visit iterate dictionary & dictionary comprehensions

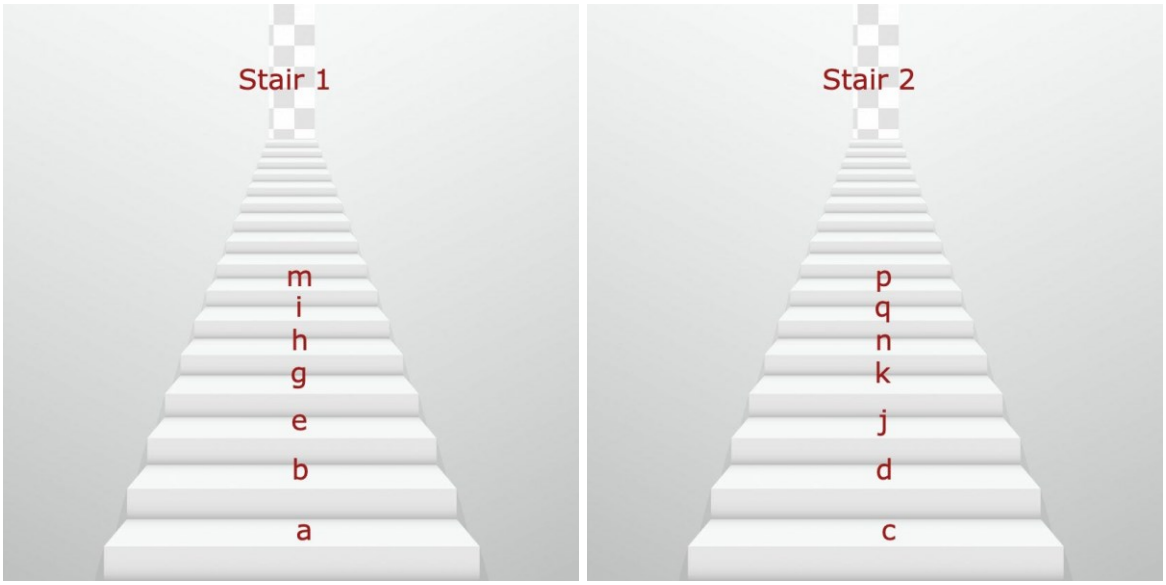
4. Uninformed Search or Blind Search

Finding for something known is called '**Search'** in contrast to what we call 'Research' which is looking for something unknown. As the name suggests the duty of AI Search algorithms is also, searching. Problem-solving is one direct application of search. Effective and efficient solutions can be formulated for real-world problems with the help of AI search algorithms.

Uninformed Search generally refers to search algorithms that have no additional information asides the one provided in the problem definition. **Breadth First Search (BFS)** is arguably the more efficient of the two search algorithms. It provides a complete, and in most cases, optimal search solution when compared to depth first search, which we will explore next.

Given an assignment, as the name implies, BFS will search the breadth at each level and proceed downwards.

Imagine we have two staircases…yeah, don't imagine, see it below



We have two stairs and we have a letter assigned to each step. If I asked you to find the letter "k", how would you find it? Keep in mind that you DO NOT know what letter is assigned to what staircase or what step. You only know that you have two staircases, and a letter is assigned to each step of the staircase.

Breadth First Search (BFS) implores us to go through both staircases one step at a time, that is, we check the first step on staircase 1 to see if the letter assigned is what we are looking for (k), if it isn't, check step 1 on staircase 2.

"Breadth First Search will check step 1 on staircase 1 and 2 before proceeding to step 2."

To find the letter k, BFS will go through a→c→b→d→e→j→g→k [we proceed from step 1 on staircase 1 and move upwards only after going to the same level on staircase 2]

Depth First Search (DFS) on the other hand suggests that we go down staircase 1 in search of k, and only search staircase 2 if we do not find it. If you are lucky, k is in staircase 1, otherwise, you'd have spent all efforts searching staircase one for no reason. NO REASON! All of that work!

Depth First Search will check staircase 1 and proceed to check staircase 2 if we do not find what we are looking for in staircase 1. That is, we are going down in depth first thus the name.”””

To find the letter k, DFS will go through a→b→e→g→h→i→m→c→d→j→k. We can see that the path to find k consists of all steps on staircase 1 before the first step on staircase 2.

5. BFS vs DFS

The problem statement and nature of information provided will determine which of the algorithms is ideal but BFS is generally more performant and efficient. **BFS is complete**, meaning if a solution exists, it will be found. It is also optimal if the cost for each step is equal to 1.

DFS on the other hand fails on completeness if the depth of the staircase is infinite or there is a loop (staircase 1 leads to another staircase, say staircase 3?). As shown in the simple example above, it is also not optimal. If staircase 1 had five thousand (5000) steps, DFS will go through all of them before checking staircase 2.

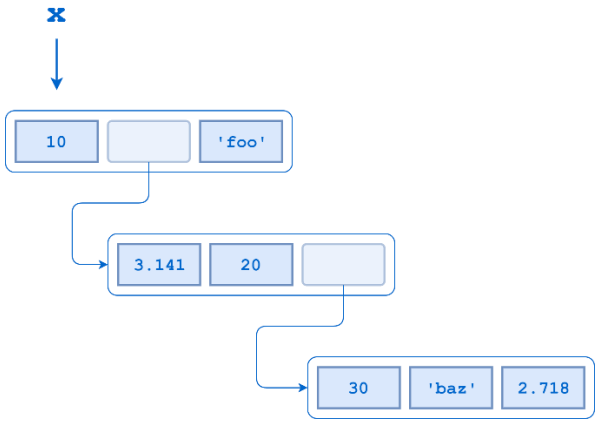
6. Exercises

6.1 Nested List (10 marks)

Consider the following nested list definition:

```
x = [10, [3.141, 20, [30, 'baz', 2.718]], 'foo']
```

A schematic for this list is shown below:



What is the expression that returns the 'z' in 'baz'?

6.2 Count Distinct Elements (10 marks)

Given a list of integers, create a function to return the count of distinct elements in the list. For example, the list [1, 2, 2, 3, 4, 2] would return 4.

6.3 Unique values in a dictionary (10 marks)

Given a dictionary, return a list of all the values in the dictionary, but without any duplicates.

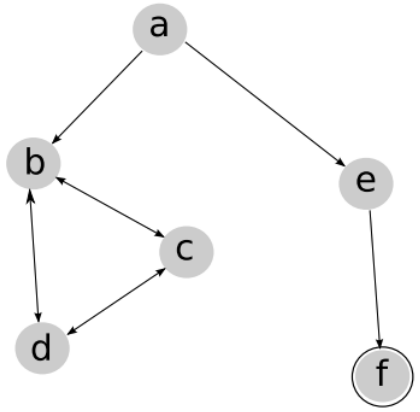
6.4 Combine dictionaries (10 marks)

Given two dictionaries d1 and d2, return a new dictionary which contains all the keys and values from d1 and d2. If a key is in both dictionaries, the value from d1 takes precedence.

6.5 DFS or BFS? (10 marks)

Consider the search problem represented in Figure below, where a is the start node and f is the goal node. Would you prefer DFS or BFS for this problem? Why?

Write python code to solve this using the selected method.



6.6 Cube Finder (50 marks)

You are tasked with developing a cube solver using blind search algorithms, specifically Depth-First Search (DFS) and Breadth-First Search (BFS). The cube is represented as a 2D array, where 0s denote open spaces and 1s represent walls. Your objective is to find a path from the starting point, located at the top left corner of the cube (0,0), to the goal point, positioned at the bottom right corner of the cube (n-1, m-1), where n and m are the dimensions of the cube.

Below are the key components of the problem:

- **Reading the cube from a text file:** Implement a function to read the cube from a text file. This function should take a filename as input and return a 2D array representing the cube. The cube is represented using a binary matrix, where 0 signifies an empty cell and 1 represents a wall.
- **Implementing DFS and BFS:** Develop functions for DFS and BFS algorithms. These functions should accept the cube as input and return a path to the goal state if one exists, or -1 if there is no path to the goal state. It is essential for students to utilize appropriate data structures and algorithms to implement DFS and BFS.
- **Returning the path to the goal state:** The function should provide a list of coordinates representing the path from the starting point to the goal state. If no path exists, the function should return -1.

Note: Multiple goal states may exist within a cube, meaning that DFS and BFS algorithms may produce different paths to reach various goal states. Consequently, the DFS algorithm may yield one path while the BFS algorithm produces another.

Here's a sample content of a text file representing the cube:

```
S 0 0 0 1 0 0
```

1 1 0 0 0 1 1
0 1 0 1 0 0 0
1 1 0 1 1 0 1
0 1 0 1 0 0 0
0 1 1 1 0 1 1
0 0 0 0 0 0 **G**

In this representation:
‘0' represents an open space.
'1' represents a wall.
Ensure that your actual cube text file follows a similar format, with appropriate dimensions and positions for the starting point ('S') and the goal point ('G').

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/