National University of Computer and Emerging Sciences

Lab Manual 02
Artificial Intelligence Lab

Instructor: Mariam Nasim

1 Python Lists

Everything is 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:

```
# ['Mercury', 'Venus', 'Earth', 'Malacandra', 'Jupiter', 'Saturn',
'Uranus', 'Neptune']
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 List Modification 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, 123,4.5]

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 comprehensions

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 Example

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. 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 in the error message. The error message usually describes the error type.

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

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.

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.")


print("This WILL be printed")
```

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

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

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

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

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.")
```

5 Exercise (30 Marks)

5.1 Check Divisible by 3 (2 Marks)

Create a function to check if a given element in a list is divisible by 3 or not. Return those elements from the function only. Give a one line solution.

Input:

list1 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Output:

```
[0, 9, 36, 81]
```

5.2 Front Back (4 Marks)

Consider dividing a string into two halves. If the length is even, the front and back halves are the same length. If the length is odd, we'll say that the extra char goes in the front half. e.g. 'abcde', the front half is 'abc', the back half 'de'. Given 2 strings, a and b, return a string of the form: a-front + b-front + a-back + b-back

Example 1:

python

```
split_and_merge("abcde", "fghij")
```

Output:

```
python

"abcfghdeij"
```

5.3 Sort tuple (3 Marks)

Given a list of non-empty tuples, return a list sorted in increasing order by the last element in each tuple.

e.g. [(1, 7), (1, 3), (3, 4, 5), (2, 2)] creates [(2, 2), (1, 3), (3, 4, 5), (1, 7)]

5.4 Multiply tuple elements (3 Marks)

Write a function which returns a tuple that is formed by multiplying adjacent elements in the tuple.

Input:

(1, 5, 7, 8, 10)

Output:

(5, 35, 56, 80)

5.5 Binary Search (5 Marks)

Given a sorted list of integers, create a function to perform binary search on the list and return the index of the target element. If the target element is not found, return -1.

Example outputs:

- For the input `[1, 2, 3, 4, 5, 6, 7]` with target `3`, the output is `2` (because 3 is at index 2).
- For the input `[1, 2, 3, 4, 5, 6, 7]` with target `6`, the output is `5` (because 6 is at index 5).
- For the input `[1, 2, 3, 4, 5, 6, 7]` with target `10`, the output is `-1` (because 10 is not in the list).

5.6 Factor count (3 Marks)

Given a list of integers, create a function to return a dictionary that contains the frequency count of each element in the list.

For example, the list [1, 2, 2, 3, 4, 2] would return {1:1, 2:3, 3:1, 4:1}.

5.7 Common values in dictionaries (4 marks)

Given a dictionary, return the key(s) of the value(s) that appear most often. If there is a tie, return all keys.

Example Outputs:

1. For the input `{'a': 1, 'b': 2, 'c': 2, 'd': 3}`, the output is `['b', 'c']` because both values 2 appear most frequently.
2. For the input `{'a': 1, 'b': 1, 'c': 1}`, the output is `['a', 'b', 'c']` because all values appear with the same frequency (1).

5.8 Find specific key in Dictionary (3marks)

Given a list of dictionaries and a value, return a new list that contains only the dictionaries that have the given value for a specified key.

- Input:

```python
list_of_dicts = [
    {'name': 'Alice', 'age': 25},
    {'name': 'Bob', 'age': 30},
    {'name': 'Charlie', 'age': 25},
    {'name': 'David', 'age': 40}
]
```

Key: `'age'` Value: `25`

- Output:

```python
[{'name': 'Alice', 'age': 25}, {'name': 'Charlie', 'age': 25}]
```

5.9 Exception handling (3)

Write a program to add, mul, divide two numbers x and y.

Implement exception handling techniques (try..except clause) for handling possible exceptions in the scenario.

Submission Instructions:

- Rename your Jupyter notebook to your "**roll number_Name**" 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.
- All outputs should be displayed properly.
- Late submissions will **NOT AT ALL** be accepted.