GET214: COMPUTING AND SOFTWARE ENGINEERING

(Programming in Python)

LESSON 5: PYTHON DATA STRUCTURES I – STRINGS & LISTS

Learning Outcomes:

At the end of the lesson, students should be able to:

1. Use string indexing and slice operator to access characters in a string
2. Use string methods - count(), find(), upper(), lower(), split(), etc
3. Iterate through strings
4. Understand lists and how to create them
5. Use list methods – append(), pop(), insert(), remove() sort(), reverse(), extend()
6. Iterate through list content of a list
7. Use of certain BIFs with list – min(), max(), sum(), len()
8. Index and slice through a list
9. Use nested loops to iterate list of lists
10. Implement filtering using list comprehension

5.1 Python Strings

A **string** is a sequence of characters. Python provides useful methods for processing string values. In previous lessons, we have learned the basics of strings and how they can be displayed using the print() function. We had also used the len() function on strings to return the number of characters in the string, including white spaces.

5.2 The String index operator []

We can access individual elements of a string using the index operator []. String indexes count from 0. Thus the first character is at index 0 while the 3rd character is at index 2.

The following code snippet and output illustrate the use of the index operator.

Python also recognizes negative indexes. The last character in a string is at index -1. Thus, negative index counts from -1 from the right while positive index counts from 0 from the left.

5.3 The Slice Operator [:]

String slicing is used when a programmer must get access to a sequence of characters. Here, a string slicing operator can be used. When [a:b] is used with the name of a string variable, a sequence of characters starting from index a (inclusive) up to index b (exclusive) is returned. Both a and b are optional. If a or b are not provided, the default values are 0 and len(string), respectively.

Example 5.1

Getting the minutes

Consider a time value is given as "hh:mm" with "hh" representing the hour and "mm" representing the minutes. To retrieve only the string's minutes portion, the following code can be used:

```
time_string = "13:46"
minutes = time_string[3:5]
print(minutes)
```

The above code's output is:

```
46
```

Consider the following code and the output and learn how the slice operator is used with the optional parameters.

```
sample = "This is GET214 - Introduction to python programming"
print(sample[8:14]) #Displays from character at index 8 to 13
print(sample[17:]) #displays from character at index 17 to the end
print(sample[:14]) #displays from the beginning to index 13
print(sample[-11:]) #displays from character at index -11 to the end
```

```
GET214
Introduction to python programming
This is GET214
programming
```

5.4 String methods

String comes with some built-in methods to help manipulate it to desired format. Table 5.1 itemizes some common string methods and their usages. The string variable **my_str**, used in table 5.1 originally has the value **"This is python programming**" stored.

Table 5.1 – Some String methods

S/N	Method	Usage	Result
1	lower()	my_str.lower()	'this is python programming'
2	upper()	my_str.upper()	'THIS IS PYTHON PROGRAMMING'
3	title()	my_str.title()	'This Is Python Programming'
4	capitalize()	my_str.capitalize()	'This is python programming'
5	count()	my_str.count('is')	2
6	find()	my_str.find('python')	8
		my_str.find('.py')	-1
7	index()	my_str.index('python')	8
		my_str.index('.py')	ValueError: substring not found
8	split()	my_str.split()	['This', 'is', 'python', 'programming']
9	replace()	my_str.replace('o','x')	'This is pythxn prxgramming'
10	isalpha()	my_str.isalpha()	False
		my_str[:4].isalpha()	True
	isdigit()	my_str.isdigit()	False
11	Join()	' '.join(['at', 'thy', 'word'])	'at thy word'

4.5 Immutability of String

String objects are **immutable** meaning that string objects cannot be modified or changed once created. Once a string object is created, the string's contents cannot be altered by directly modifying individual characters or elements within the string.

Instead, to make changes to a string, a new string object with the desired changes is created, leaving the original string unchanged.

Example 5.1: Getting the time's minute portion

Consider a time value is given as part of a string using the format of `"hh:mm"` with `"hh"` representing the hour and `"mm"` representing the minutes. To retrieve only the string's minute portion, the following code can be used:

```
time_string = "The time is 12:50"
index = time_string.index(":")
print(time_string[index+1:index+3])
```

The above code's output is:

```
50
```

Activity 5.1: Changing the greeting message

Given the string "Hello my fellow classmates" containing a greeting message, print the first word by getting the beginning of the string up to (and including) the 5th character. Change the first word in the string to "Hi" instead of "hello" and print the greeting message again.

Activity 5.2: Finding all Spaces

Write a program that, given a string, counts the number of space characters in the string. Also, print the given string with all spaces removed.

```
Input: "This is great"
```

```
prints:
2
Thisisgreat
```

Activity 5.3: Unique and comma separated words

Write a program that accepts a comma-separated sequence of words as input, and prints words in separate lines. Ex: Given the string `"happy,smiling,face"`, the output would be:

```
happy
smiling
face
```

5.6 Introduction to Python Lists

Programmers often work on collections of data. Lists are a useful way of collecting data elements. Python lists are extremely flexible, and, unlike strings, a list's contents can be changed. List is therefore a mutable data structure.

5.7 Using lists methods the modify lists

- An **append**() operation is used to add an element to the end of a list. In programming, **append** means add to the end.
- A **remove**() operation removes the specified element from a list.
- A **pop**() operation removes the last item of a list.
- The **insert**() method adds element to the list at the specified index positon.

Example 5.2: Simple operations to modify a list

The code below demonstrates simple operations for modifying a list. Line 8 shows the append() operation, line 12 shows the remove() operation, and line 17 shows the pop() operation. Since the pop() operation removes the last element, no parameter is needed.

```
"""Operations for adding and removing elements from a list."""

    # Create a list of students working on a project
    student_list = ["Jamie", "Vicky", "DeShawn", "Tae"]
    print(student_list)

    # Another student joins the project. The student must be added
to the list.
    student_list.append("Ming")
    print(student_list)

    # "Jamie" withdraws from the project. Jamie must be removed
from the list.
    student_list.remove("Jamie")
    print(student_list)

    # Suppose "Ming" had to be removed from the list.
    # A pop() operation can be used since Ming is last in the
list.
    student_list.pop()
    print(student_list)
```

The above code's output is:

```
['Jamie', 'Vicky', 'DeShawn', 'Tae']
['Jamie', 'Vicky', 'DeShawn', 'Tae', 'Ming']
['Vicky', 'DeShawn', 'Tae', 'Ming']
['Vicky', 'DeShawn', 'Tae']
```

5.7 Iterating a List

An iterative for loop can be used to iterate through a list. Alternatively, lists can be iterated using list indexes with a counting for loop.

Given the following assignment,

```
names = ['Jack', 'Mark', 'Mary', 'Glory']
```

We ca iterate through the list using the list index thus:

```
for i in range(len(names)):
    print(names[i])
```

```
Jack
Mark
Mary
Glory
```

We can also iterate through the list by naming the items of the list instead of using indexes.

```
for name in names:
    print(name)
```

```
Jack
Mark
Mary
Glory
```

5.8 Sorting a list

Ordering elements in a sequence is often useful. Sorting is the task of arranging elements in a sequence in ascending or descending order.

Sorting can work on numerical or non-numerical data. When ordering text, dictionary order is used. Ex: "bat" comes before "cat" because "b" comes before "c".

Python provides methods for arranging elements in a list.

- The sort() method arranges the elements of a list in ascending order. For strings, ASCII values are used and uppercase characters come before lowercase characters, leading to unexpected results. Ex: "A" is ordered before "a" in ascending order but so is "G"; thus, "Gail" comes before "apple".
- The reverse() method reverses the elements in a list.

5.8 Common List Operations

The **max()** function called on a list returns the largest element in the list. The **min()** function called on a list returns the smallest element in the list. The max() and min() functions work for lists as long as elements within the list are comparable.

The **sum()** function called on a list of numbers returns the sum of all elements in the list.

5.9 Indexing and slicing through a list

We can access individual elements of a list using its index as illustrated in section 5.7. As strings, list indexes count from 0 from the left or -1 from the right.

Given the assignment, `scores = [48, 73, 21, 65, 48, 57, 63, 80]`, The following illustrate the indexing and slicing operations:

- The third item on the list, 21, is accessed through `scores[2]`
- The last item, 80 is accessed by `scores[-1]`
- The first 3 items are access by `scores[:4]`
- The last 3 items are accessed by `scores[-3:]`

5.10 Nested List

Lists can be made of any type of element. A list element can also be a list. Example: [2, [3, 5], 17] is a valid list with the list [3, 5] being the element at index 1.

When a list is an element inside a larger list, it is called a nested list. Nested lists are useful for expressing multidimensional data. When each of the elements of a larger list is a smaller list, the larger list is called a list-of-lists.

We use a double index operator to access elements of an inner list. For instance, given that the `siwes_score = [1, [4,8,2], [5,3], 6]`, to get the item 5 from the list [5,3], we use `siwes_score[2][0]`.

5.11 List Comprehension

A list comprehension is a Python statement to compactly create a new list using a pattern.

The general form of a list comprehension statement is shown below.

```
list_name = [expression for loop_variable in iterable]
```

list_name refers to the name of a new list, which can be anything, and the for is the for loop keyword. An expression defines what will become part of the new list. loop_variable is an iterator, and iterable is an object that can be iterated, such as a list or string.

Example 5.3: Creating a new list with a list comprehension

A list comprehension shown below in the second code has the same effect as the regular for loop shown in the first code. The resultant list is `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]` in both cases.

Creating a list of squares using a for loop.

```
# Create an empty List.

squares_list = []

# Add items to a list, as squares of numbers starting at
#0 and ending at 9.


for i in range(10):
    squares_list.append(i*i)
```

Creating a list of squares using the list comprehension.

```
square_list = [i*i for i in range(10)]
```

The expression i*i is applied for each value of the loop_variable i.

5.12 Filtering Using List Comprehension

List comprehensions can be used to filter items from a given list. A condition is added to the list comprehension.

```
list_name = [expression for loop_variable in container if condition]
```

In a filter list comprehension, an element is added into list_name only if the condition is met.

```
odd_nums = [x for x in range(20) if x%2 == 1]
print(odd_nums)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Concepts in Practice Questions to attempt from the textbook

Pages 119 – 209, 224 – 230