Activity No. 4.2 Strings	
Course Code: CPE 103	**Program:** Computer Engineering
Course Title: Object Oriented Programming	**Date Performed:** 04/12/25
Section: 1-A	**Date Submitted:** 04/12/25
Name: Villanueva, Bryan 0.	**Instructor:** Engr. Maria Rizette H. Sayo

1. Objective(s)

This activity aims to demonstrate the students' understanding of strings implementation in solving problems.

2. Intended Learning Outcomes (ILOs)

After this module, the students should be:

- Demonstrate the implementation of strings in different use cases and acquire skills in solving difficult problems using strings.

3. Discussion

A string is a sequence

A string is a *sequence* of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement extracts the character at index position 1 from the fruit variable and assigns it to the letter variable.

The expression in brackets is called an *index*. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> print(letter)
a
```

For most people, the first letter of “banana” is “b”, not “a”. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

So “b” is the 0th letter (“zero-th”) of “banana”, “a” is the 1th letter (“one-th”), and “n” is the 2th (“two-th”) letter.

String Indexes

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

Getting the length of a string using len

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
```

```
>>> last = fruit[length]
IndexError: string index out of range
```
The reason for the IndexError is that there is no letter in “banana” with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:
```
>>> last = fruit[length-1]
>>> print(last)
a
```
Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

Traversal through a string with a loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a *traversal*. One way to write a traversal is with a while loop:
```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```
This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

Another way to write a traversal is with a for loop:
```
for char in fruit:
    print(char)
```
Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

String slices

A segment of a string is called a *slice*. Selecting a slice is similar to selecting a character:
```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```
The operator [n:m] returns the part of the string from the “n-th” character to the “m-th” character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:
```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an *empty string*, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Strings are immutable

It is tempting to use the operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an *object* is the same thing as a value, but we will refine that definition later. An *item* is one of the values in a sequence.

The reason for the error is that strings are *immutable*, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

Looping and counting

The following program counts the number of times the letter "a" appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a *counter*. The variable count is initialized to 0 and then incremented each time an "a" is found. When the loop exits, count contains the result: the total number of a's.

The in operator

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

String methods

Strings are an example of Python *objects*. An object contains both data (the actual string itself) and *methods*, which are effectively functions that are built into the object and are available to any *instance* of the object.

Python has a function called dir which lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
[... 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(self, /)
    Return a capitalized version of the string.

    More specifically, make the first character have upper
    case and the rest lower case.
>>>
```

While the dir function lists the methods, and you can use help to get some simple documentation on a method.

Calling a *method* is similar to calling a function (it takes arguments and returns a value) but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.
For example, the method upper takes a string and returns a new string with all uppercase letters:
Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.
A method call is called an *invocation*; in this case, we would say that we are invoking upper on the word.
For example, there is a string method named find that searches for the position of one string within another:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter.
The find method can find substrings as well as characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the strip method:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Some methods such as *startswith* return boolean values.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

You will note that startswith requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the lower method.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
```

```
>>> line.lower().startswith('h')
True
```
In the last example, the method lower is called and then we use startswith to see if the resulting lowercase string starts with the letter “h”. As long as we are careful with the order, we can make multiple method calls in a single expression.

Parsing strings

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

From stephen.marquard@ *uct.ac.za* Sat Jan 5 09:14:16 2008

and we wanted to pull out only the second half of the address (i.e., uct.ac.za) from each line, we can do this by using the find method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space *after* the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```
We use a version of the find method which allows us to specify a position in the string where we want find to start looking. When we slice, we extract the characters from “one beyond the at-sign through up to *but not including* the space character”.

Formatted String Literals

A formatted string literal (often referred to simply as an f-string) allows Python expressions to be used within string literals. This is accomplished by prepending an f to the string literal and enclosing expressions in curly braces {}.

For example, wrapping a variable name in curly braces inside an f-string will cause it to be replaced by its value:

```
>>> camels = 42
>>> f'{camels}'
'42'
```
The result is the string ‘42’, which is not to be confused with the integer value 42.

An expression can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> f'I have spotted {camels} camels.'
'I have spotted 42 camels.'
```
Several expressions can be included within a single string literal in order to create more complex strings.

```
>>> years = 3
```

```
>>> count = .1
>>> species = 'camels'
>>> f'In {years} years I have spotted {count} {species}.'
'In 3 years I have spotted 0.1 camels.'
```

Reference:
PY4E - Python for everybody. (n.d.). https://www.py4e.com/html3/06-strings

4. Materials and Equipment

To properly perform this activity, the student must have:

- Python
- Spyder IDE
- Jupyter Notebook

5. Procedure

1. Open the Anaconda.
2. Use the jupyter notebook and follow the instructions below.
3. Provide a screenshot for every test case in each code and insert in the Output section with a corresponding description and observation.

A string is a sequence

```
[ ]  fruit = 'banana'
     letter = fruit[1]
```

```
[ ]  print(letter)
```

```
[ ]  letter = fruit[0]
     print(letter)
```

```
[ ]  letter = fruit[1.5]
```

Getting the length of a string using len

```
[ ]  fruit = 'banana'
     len(fruit)
```

```
[ ]  length = len(fruit)
     last = fruit[length]
```

```
[ ]  last = fruit[length-1]
     print(last)
```

Traversal through a string with a loop

```
[ ]  index = 0
     while index < len(fruit):
         letter = fruit[index]
         print(letter)
         index = index + 1
```

```
[ ]  for char in fruit:
         print(char)
```

String slices

```
[ ]  s = 'Monty Python'
     print(s[0:5])
```

```
[ ]  print(s[6:12])
```

```
[ ]  fruit = 'banana'
     fruit[:3]
```

```
[ ]  fruit[3:]
```

```
[ ]  fruit = 'banana'
     fruit[3:3]
```

Strings are immutable

```
[ ]  greeting = 'Hello, world!'
     greeting[0] = 'J'
```

```
[ ]  greeting = 'Hello, world!'
     new_greeting = 'J' + greeting[1:]
     print(new_greeting)
```

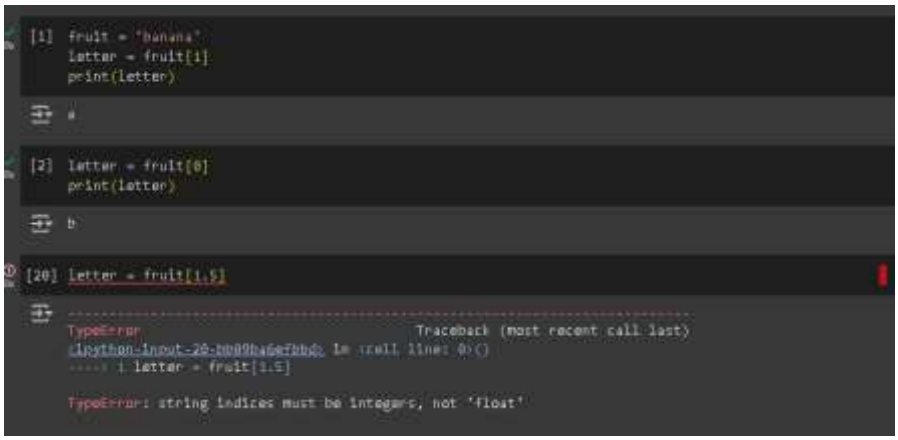
Looping and counting

```
[ ]  word = 'banana'
     count = 0
     for letter in word:
         if letter == 'a':
             count = count + 1
     print(count)
```

6. Output

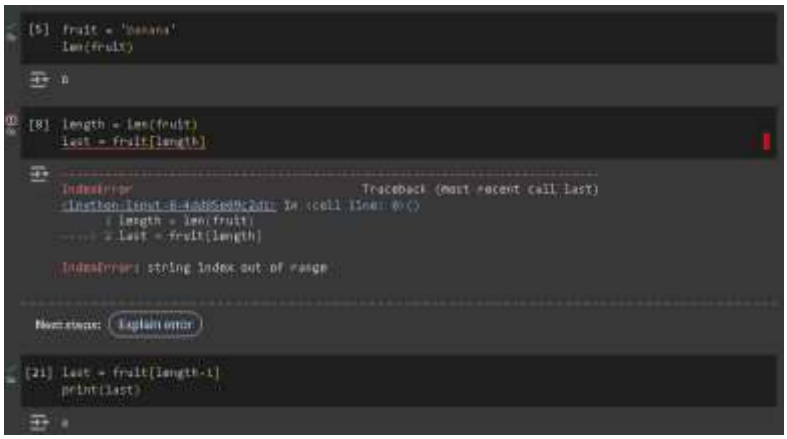
Provide an output of your work here. (include an analyzation for every screenshot or output)

A string is a sequence



- The code tries to access characters from the string 'banana' using indexing. It first gets 'a' at index 1, then 'b' at index 0. But it fails when trying to use 1.5 as an index because string indices must be integers.

Getting the length of a string using len



- The code uses the string `'banana'` and shows how indexing works. It calculates the length as `6` but fails when trying to access the last character using `length` instead of `length-1`. Fixing it prints `'a'`, showing why proper indexing is important.

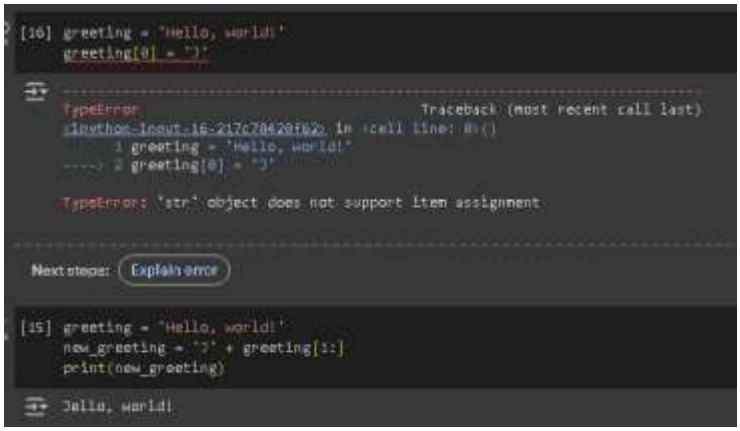
Traversal through a string with a loop

- The code shows two ways to loop through a string in Python. The first example uses a “while” loop to print each character one by one, based on the index. The second example uses a “for” loop to print each character directly from the string. Both methods display "banana" character by character and work as expected.

String slices

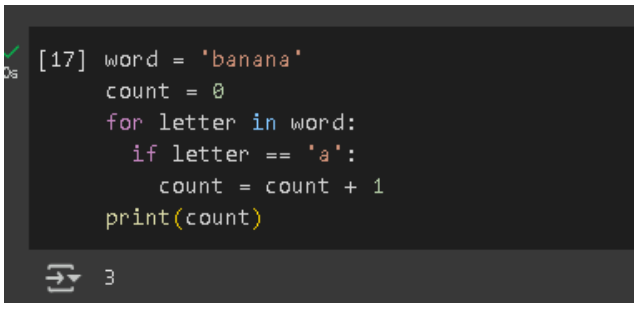
- The code shows how to slice strings in Python. It extracts "Monty" and "Python" from "Monty Python" using specific indices. It slices "banana" to get "ban" with [:3] and "ana" with [3:]. Slicing "banana" with [3:3] gives an empty string because the start and end are the same. This explains how slicing works.

Strings are immutable



- The code shows two ways to change a string. The first way tries to modify it directly, but it gives an error because strings can't be changed. The second way creates a new string using slicing, which works and outputs `'Jello, world!'`. This shows how to handle strings in Python.

Looping and counting



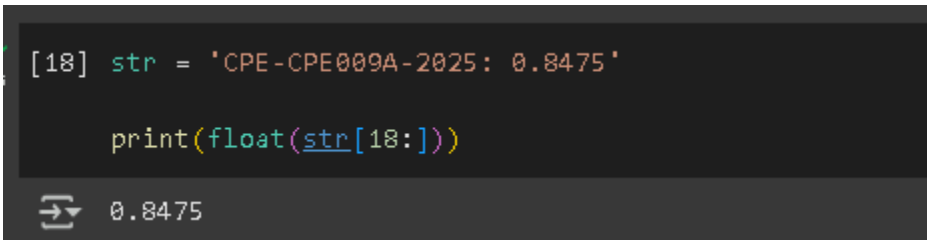
- The code counts how many times 'a' appears in 'banana'. It starts with count = 0, goes through each letter, and adds to count whenever it finds 'a'. The final result is 3. This shows how to use loops to check strings.

7. Supplementary Activity

Solve the following problems:

1. Take the following Python code that stores a string:
   str = 'CPE-CPE009A-2025: 0.8475'
   Use find and string slicing to extract the portion of the string after the colon character and then use the float function to convert the extracted string into a floating point number.

*Insert the code and the output here with the corresponding analyzation.

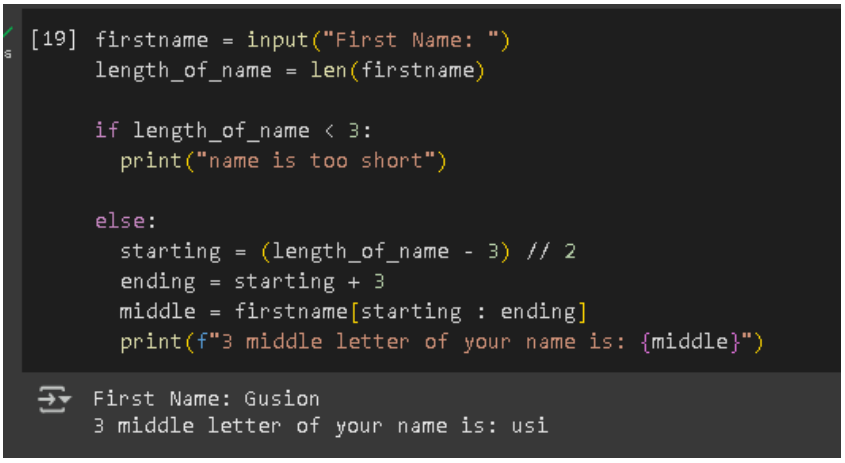


```
[18] str = 'CPE-CPE009A-2025: 0.8475'

     print(float(str[18:]))

     0.8475
```

- The code takes a string 'CPE-CPE000A-2025: 0.8475' and extracts a part of it starting from the 18th character. It converts this part to a floating point number and prints it. The output is 0.8475, showing how to extract and work with specific parts of a string.

2. Write a python program that asks the user for their first name and convert it in to as a string. Print out the 3 words middle of the name using slice.

*Insert the code and the output here with the corresponding analyzation.



```
[19] firstname = input("First Name: ")
     length_of_name = len(firstname)

     if length_of_name < 3:
       print("name is too short")

     else:
       starting = (length_of_name - 3) // 2
       ending = starting + 3
       middle = firstname[starting : ending]
       print(f"3 middle letter of your name is: {middle}")

     First Name: Gusion
     3 middle letter of your name is: usi
```

- The code checks the length of a name entered by the user. If the name has less than 3 letters, it says the name is too short. If the name is longer, it finds and prints the three middle letters using slicing. For "Gusion," the result is "usi."

8. Conclusions/Observations

This activity helped me understand how to use strings to solve different types of problems. By working through the exercises, I learned how to apply string operations in practical ways and gained more confidence in handling complex tasks. Overall, it showed me how important strings are in programming and how they can be used to solve challenging problems effectively.

9. Assessment Rubric