|  |  |
| --- | --- |
| **Activity No. 4.2 Strings** | |
| **Course Code:** CPE 103 | **Program:** Computer Engineering |
| **Course Title:** Object Oriented Programming | **Date Performed**: 04/12/25 |
| **Section:** BSCpE 1A | **Date Submitted:** 04/12/25 |
| **Name:** Nerio, Hannah Grace A. | **Instructor:** Engr. Maria Rizette H.Sayo |
| 1. **Objective(s)** | |
| This activity aims to demonstrate the students' understanding of strings implementation in solving problems. | |
| 1. **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. | |
| 1. **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* | |
| 1. **Materials and Equipment** | |
| To properly perform this activity, the student must have:   * Python * Spyder IDE * Jupyter Notebook | |
| 1. **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. | |
| 1. **Output** | |
| Provide an output of your work here. (include an analyzation for every screenshot or output)    This code shows how strings in Python are sequences, meaning you can access each character using an index. In the first two examples, fruit[1] and fruit[0] correctly return 'a' and 'b' because they use whole numbers (integers) as indices. However, the last example uses fruit[1.5], which causes an error. That’s because Python only allows integers for indexing strings—not decimals (floats). To fix the error, you should use a whole number like 1 instead of 1.5.p    This code is meant to show how we can get specific letters from a string by using their position, or index. In Python, strings work like a list of characters, and we can get each character by using square brackets with an index number. The string 'banana' is used, and in the first two lines, the code runs successfully because it uses valid whole numbers—1 and 0—to get characters from the string. However, in the last line, the code uses 1.5 as the index, which causes an error. This is because Python does not allow decimal numbers (also called floats) when trying to get a character from a string. It only accepts whole numbers, or integers, for indexing. Since 1.5 is not a valid index, Python shows a TypeError, saying that string indices must be integers, not floats. To fix the error, the code should use a whole number like 1 instead of 1.5.  A screenshot of a computer program  AI-generated content may be incorrect.  This code shows how to go through a string and use slicing. The first two blocks print each letter in 'banana' using a while loop and a for loop—they both work the same way. The last part uses slicing to get parts of the string 'Monty Python' and 'banana'. Slices like fruit[:3] give 'ban', but fruit[3:1] and fruit[3:3] return empty strings. That’s because the start index is greater than or equal to the end index, so there’s nothing to show. The code runs fine—no errors—just empty results in those cases.    This code shows that strings in Python are immutable, meaning you can't change them directly. The first line tries to change the first letter of 'Hello, world!' to 'J', but it causes an error because strings don’t allow item assignment. The second part works by creating a new string: it combines 'J' with the rest of the original string using slicing. This way, you get 'Jello, world!' without changing the original string.    This code loops through the word 'banana' and counts how many times the letter 'a' appears. It starts with count = 0, then checks each letter in the word. If the letter is 'a', it adds 1 to the count. After the loop, it prints the total, which is 3. The code runs fine, and there’s no error. Everything works as expected. | |
| 1. **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.  A screenshot of a computer  AI-generated content may be incorrect.  In this activity, the goal is to extract the number found after the colon in the string 'CPE-CPE009A-2025: 0.8475'. The code uses the find() function to locate the position of the colon and then slices the string starting two characters after it (to skip the colon and the space). The sliced part is a string, so it is converted into a float using the float() function. The final output is the number 0.8475. This kind of code is helpful when working with formatted text, such as logs or reports, where specific values need to be pulled out of strings. |  1. 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.  A screenshot of a computer  AI-generated content may be incorrect.  In this the user ask to enter their first name and then prints out the three letters from the middle of that name. The program first checks if the name has at least three characters. If it does, it calculates the correct starting index for slicing the middle part using a basic formula. Then, it uses string slicing to extract three characters starting from the middle and prints them. If the name is too short, it tells the user. This activity is useful for understanding how string length and slicing work in Python. It can be applied to various situations like creating short usernames, nicknames, or even formatting names for display | | |
| 1. **Conclusions/Observations** | |
| During this lab, I encountered some common errors while working with strings. I learned that string indexing must use whole numbers (integers), and that trying to use decimal indexes or indexes outside the range of the string causes errors. I also learned that strings are immutable, which means I cannot change a letter directly—I have to build a new string instead. One of the most helpful lessons was understanding how string methods like .find() and .upper() work, and how they must be written with the dot notation. | |
| 1. **Assessment Rubric** | |
|  | |