Lab 01: Functions, Conditionals, Loops

Total Points: 10 (Correctness) + 1 extra credit (optional)

Due: Tuesday, January 16th, 11:59pm

Extra Credit Opportunity

You will have the opportunity to receive a **1 point** extra credit on that lab if you submit your last attempt early (refer to this section of each lab for the early submission deadline). Note: Each lab is graded out of 10 points, and you could possibly have more than 10 points in one lab.

Early Submission Date (lab01): Thursday, January 11th, 11:59pm (SD time)

Preview

Unlike DSC10, we will not be using any imported packages in the course. We have a diverse background in terms of programming experience. Therefore, we will be having a long but useful lab01 that helps you get used to Python built-in methods and some structures you can use in future assignments! Please read every single word in this lab! If you have any questions, please ask on *ED Discussion or come to chat with our tutors during Office Hours*!

Starter Files

Download lab01.zip. Inside the archive, you will find starter files for the questions in this lab. You can't import anything to solve the problems. To unzip a file, simply just double-click on the archive in the download folder, and it should populate a lab01.py file. Move the lab01.py to your DSC20 folder manually.

Tip: You can comment out all questions first using "Ctrl + /" (Windows) or "Cmd + /" (Mac). Once you finish one question, you can uncomment a new one. This way doctests will give you a cleaner output.

Submission

By the end of this lab, you should submit the lab via Gradescope. You may submit more than once before the deadline; only the final submission will be graded.

Logistics

Using Python

When running a Python file, you can use the **options** on the command line to inspect your code further. Here are a few that will come in handy. If you want to learn more about other Python command-line options, take a look at the documentation.

- **Disclaimer**: If you are using Windows, you may need to type `py`, rather than `python3`
- Using no command-line options will run the code in the file you provide and return you to the command line.
  ```
  >>> python3 lab01.py
  ```
- **-i**: The `-i` option runs your Python script, then opens an interactive session. In an interactive session, you run Python code line by line and get immediate feedback instead of running an entire file all at once.
- To exit, type `exit()` into the interpreter prompt. You can also use the keyboard shortcut `Ctrl-D` on Linux/Mac machines or `Ctrl-Z Enter` on Windows.
  If you edit the Python file while running it interactively, you will need to **exit** and **restart** the interpreter in order for those changes to take effect.

  ```
  >>> python3 -i lab01.py

  >>> concat_str_1(['Welcome', 'to', 'DSC', '20!'])
  'Welcome to DSC 20!'
  ```

  - Using the interactive mode allows you to test individual doctests and see their output, but you have to type or paste tests one at a time.

- **-m doctest**: Runs all doctests in a particular file. Doctests are surrounded by triple quotes (`"""`) within functions. Each test consists of `>>>` followed by some Python code and the expected output. We have provided doctests for you in the starter code, and you may add additional doctests if you wish.
  ```
  >>> python3 -m doctest lab01.py
  ```

`return` and `print` inside a function

Most functions that you define will contain a `return` statement. The `return` statement will give the result of some computation back to the caller of the function and exit the function. For example, the function `square` below takes in a number `x` and *returns* its square.

```
def square(x):
    """
    >>> square(4)
    16
```

```
    """
    return x * x
```

When Python executes a return statement, the function terminates *immediately*. If Python reaches the end of the function body without executing a return statement, it will automatically return None.

In contrast, the print function is used to display values in the Terminal. This can lead to some confusion between print and return because calling a function in the Python interpreter will print out the function's return value.

However, unlike a return statement, when Python evaluates a print expression, the function does *not* terminate immediately.

```
def what_prints():
    print('Hello World!')
    return 'Exiting this function.'
    print('DSC20 is awesome!')

>>> what_prints()
Hello World!
'Exiting this function.'
```

Notice also that print will display text **without the quotes**, but return will preserve the quotes. We will discuss the reason later in the quarter.

Error Messages

By now, you've probably seen a couple of error messages. They might look intimidating, but error messages are *very* helpful for debugging code. The following are some common types of errors:

Error Types	Descriptions
`SyntaxError`	Contained improper syntax (e.g. missing a colon after an if statement or forgetting to close parentheses/quotes)
`IndentationError`	Contained improper indentation (e.g. inconsistent indentation of a function body)
`TypeError`	Attempted operation on incompatible types (e.g. trying to add a function and a number) or called function with the wrong number of arguments

`ZeroDivisionError`	Attempted division by zero

Using these descriptions of error messages, you should be able to get a better idea of what went wrong with your code. **If you run into error messages, try to identify the problem before asking for help.** You can often Google unfamiliar error messages to see if others have made similar mistakes to help you debug.

For example:

```
>>> square(3, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: square() takes 1 positional argument but 2 were given
```

Note:

- The last line of an error message tells us the type of the error. In the example above, we have a `TypeError`.
- The error message tells us what we did wrong -- we gave `square` 2 arguments when it can only take in 1 argument. In general, the last line is the most helpful.
- The second to last line of the error message tells us on which line the error occurred. This helps us track down the error. In the example above, `TypeError` occurred at `line 1`.

General Requirements:

- **DO NOT IMPORT ANY PACKAGES.** This includes numpy, babypandas, etc. and this applies to ALL assignments in this class (unless specifically instructed)
- Unless otherwise noted, you may use any Python built-in function if you know how it works. If you're not familiar with any function, the following are your best friend:
    1. Python's official documentation;
    2. Google: StackOverflow, w3schools, Programiz, RealPython, etc.
    3. If you're still confused, you may make an ED post.
    4. Avoid using chatGPT, it often gives wrong or overcomplicated answers.

Required Questions

This quarter we will take a trip into a magic world called Pythonia. You can expect different challenges and traps along the way. Try to beat them all and survive your

journey! Unfortunately, it is not easy to find an entrance to the magic land so we need to start looking.

Question 1:

You find yourself in the middle of the field but you have no idea where to go next. Write a function, *direction*, that takes one parameter *choice* (as an integer). Your function returns different strings depending on the parameter.

- If *choice* is positive and even, return "going east"
- If *choice* is negative and even, return "going south"
- If *choice* is positive and odd, return "going west"
- If *choice* is negative and odd, return "going north"
- Otherwise return "staying"

```
def direction(choice):
    """
    >>> direction(10)
    'going east'
    >>> direction(-10)
    'going south'
    >>> direction(5)
    'going west'
    >>> direction(-7)
    'going north'
    >>> direction(0)
    'staying'
    """
```

Question 2:

You decide to walk towards the chosen direction but you do not know how many steps to take. Write a function, *steps*, that takes two parameters: a list of positive integers and a positive integer, *target*. Your function should return the number of occurrences of the *target* in the given list. You can assume that the input list is never empty.

```
def steps(lst, target):
    """
    >>> steps([1,2,3], 2)
    1
    >>> steps([1, 2, 1, 3, 1, 4], 1)
    3
```

```
    >>> steps([1, 2, 1], 1)
    2
    """
```

Question 3:

After walking for a few miles you realized that you have seen this place before. Write a function that takes a non-empty list and returns True if the first element is the same as the last element and False otherwise.

```
def same_ends(lst):
    """
    >>> same_ends([1, 3, 1])
    True
    >>> same_ends([4, 7, 4.0])
    True
    >>> same_ends([1, 5, 4, 5])
    False
    >>> same_ends([1])
    True
    >>> same_ends(["search", "search", "Search"])
    False
    """
```

Question 4:

Now you are completely lost, because regardless of the direction, you always come back. Write a function that takes a list of **non-empty** lists and returns the number of lists that have the first and last elements the same. You should reuse your solution from the previous question by calling the `same_ends` function.

```
def quantity_of_same_ends(lsts):
   """
   >>> quantity_of_same_ends([[1, 3, 1],[1, 3, 12]])
   1
   >>> quantity_of_same_ends([[1, 3, 1],[4, 7, 4.0]])
   2
   >>> quantity_of_same_ends([[1, 3, 1],[4, 7, 4.0], [1], [3,4]])
   3
   """
```

Question 5:

You decided to just follow one direction and see where it takes you. To be safe, you want your friends to know where you are heading. Write a function that takes two parameters: a string, `name` (a name of a friend) and an integer, *choice*. Note that `name` could be an empty string – that is how missing data is created sometimes. Depending on the value of the integer, your function should return a message to your friend.

Notes:

- The return result for the values of the variable *choice* is the same as in Problem 1.
- Call function `direction` in your solution to this problem. Do not copy/paste the code from the first problem.

```
def message(name, choice):
    """
    >>> message("Marina", 5)
    'Dear Marina, I am going west.'
    >>> message("Anna", -5)
    'Dear Anna, I am going north.'
    >>> message("Rain", 0)
    'Dear Rain, I am staying.'
    """
```

Question 6:

You need to track your steps so that your friends can find you if needed. Write a function `steps_unfold` that takes one parameter: non negative integer, *steps*. Your function should return a list of integers from 1 up to *steps*, inclusively.

Note:

- First, you need to declare an empty list.
- In order to add values to a list, use function append. Here is a link to the documentation: link

```
def steps_unfold(steps):
    """
    >>> steps_unfold(1)
    [1]
    >>> steps_unfold(5)
    [1, 2, 3, 4, 5]
    >>> steps_unfold(0)
```

```
    []
    """
```

Question 7:

You realized that sending one message at the time is not time efficient, so you decided to send messages in bulk. Write a function, message_to_all, that takes two parameters: a list of strings, *names*, and an integer, *choice*. Your function should return a list of strings with messages to your friends (same order as the input names). If the list is empty, then return an empty list.

Notes:

- The return result for the values of the variable *choice* is the same as in Problem 1.
- What function should you call in your code to make your solution short? Do not copy/paste the code!

```
def message_to_all(names, choice):
    """
    >>> names = ["Marina", "Ben", "Bryce"]
    >>> message_to_all(names, 5)
    ['Dear Marina, I am going west.', \
'Dear Ben, I am going west.', \
'Dear Bryce, I am going west.']

    >>> names = ["Anastasiya", "Teresa"]
    >>> message_to_all(names, 0)
    ['Dear Anastasiya, I am staying.', \
'Dear Teresa, I am staying.']

    >>> names = []
    >>> message_to_all(names, 314)
    []
    """
```

Question 8:

It is time to take a break from a long walk. You noticed a lot of little pieces of bark laying around…what if you put them all together?...

Question 8.1:

Write a function called `combine_words_no_separator` that takes one parameter: a list of words. It returns a string that concatenates these words. If the list of words is empty, return an empty string.

Note: In DSC 20, you have to learn how to read Python's documentation on built-in functions. For example, your task is to convert a list of words (as strings) into a sentence, where there should be (or not) a separator added in between each pair of words. If the list is empty, you should return an empty string. (Notice the given doctest doesn't test on this edge case, therefore it is always recommended to add your own doctests to test for edge cases as the given doctests are always not sufficient!)

Hint: There are two usual ways to do this:

- One is through string concatenation, which starts with an empty string, and concatenates (add) each string in the list together.
- Another way to achieve the same result is to use the `join()` function.
- Note, that for 8.1 the words are not separated.

```
def combine_words_no_separator(words):
    """
    >>> combine_words_no_separator(["Very", "close"])
    'Veryclose'
    >>> combine_words_no_separator(["What", "is", "your", "name?"])
    'Whatisyourname?'
    >>> combine_words_no_separator(["We", "need", "to",  "talk"])
    'Weneedtotalk'
    """
```

Question 8.2:

We have to agree that the output is not very readable, it would be better to separate the words from each other. Write a function called `combine_words_with_separator` that takes two parameters: a list of words and a separator (as a string). Then it returns a string that concatenates the words using the separator. If the list of words is empty, return an empty string.

Note: use `.join` for a one line solution.

```
def combine_words_with_separator(words, separator):
    """
    >>> combine_words_with_separator(["Very", "close"], "***")
```

```
    'Very***close'
    >>> combine_words_with_separator(["No", "need", "to", "go"], "-")
    'No-need-to-go'
     >>> combine_words_with_separator(["Find", "another", "message"], "
")
    'Find another message'
    """
```

Question 9:

After you found the correct location of the entrance you weren't tired any more and started looking for a message. Eventually you found it pinned to one of the trees: this message has something to do with your search! Write a function called `getting_hint`, that takes one parameter: a string. Your function should return the sum of the lengths of the first and last word in this string. If the string is empty, then return a 0.

Note: There is another very handy method: `.split()`. It allows you to split a string into a list. Here is a link to the example: link

```
def getting_hint(message):
   """
    >>> getting_hint("Run away from here")
    7
    >>> getting_hint("DANGER!!")
    16
    >>> getting_hint("")
    0
    >>> getting_hint("number of steps")
    11
    """
```

Question 10:

You flipped the paper and found a few lines of instructions: walk to the left, walk straight but the total number of steps can be more than the sum of the lengths of the first and last word of the message. Write a function called `digging` that takes in three parameters.

- `steps_left`: the number of steps to move left, represented as a string.
- `steps_forward`: the number of steps to move forward, represented as a string.
- `message`: the message pinned to the tree, represented as a string.

Your function should return a string:

- 'Too far' if the sum of the steps to the left and forward is **greater** than the sum of the lengths of the first and last words of the message (the third parameter).
- 'Too close' if the sum of the steps to the left and forward is **less** than the sum of the lengths of the first and last words of the message (the third parameter).
- 'Dig here!' if the sum of the steps to the left and forward is **exactly the same** as the sum of the lengths of the first and last words of the message (the third parameter).

For example,

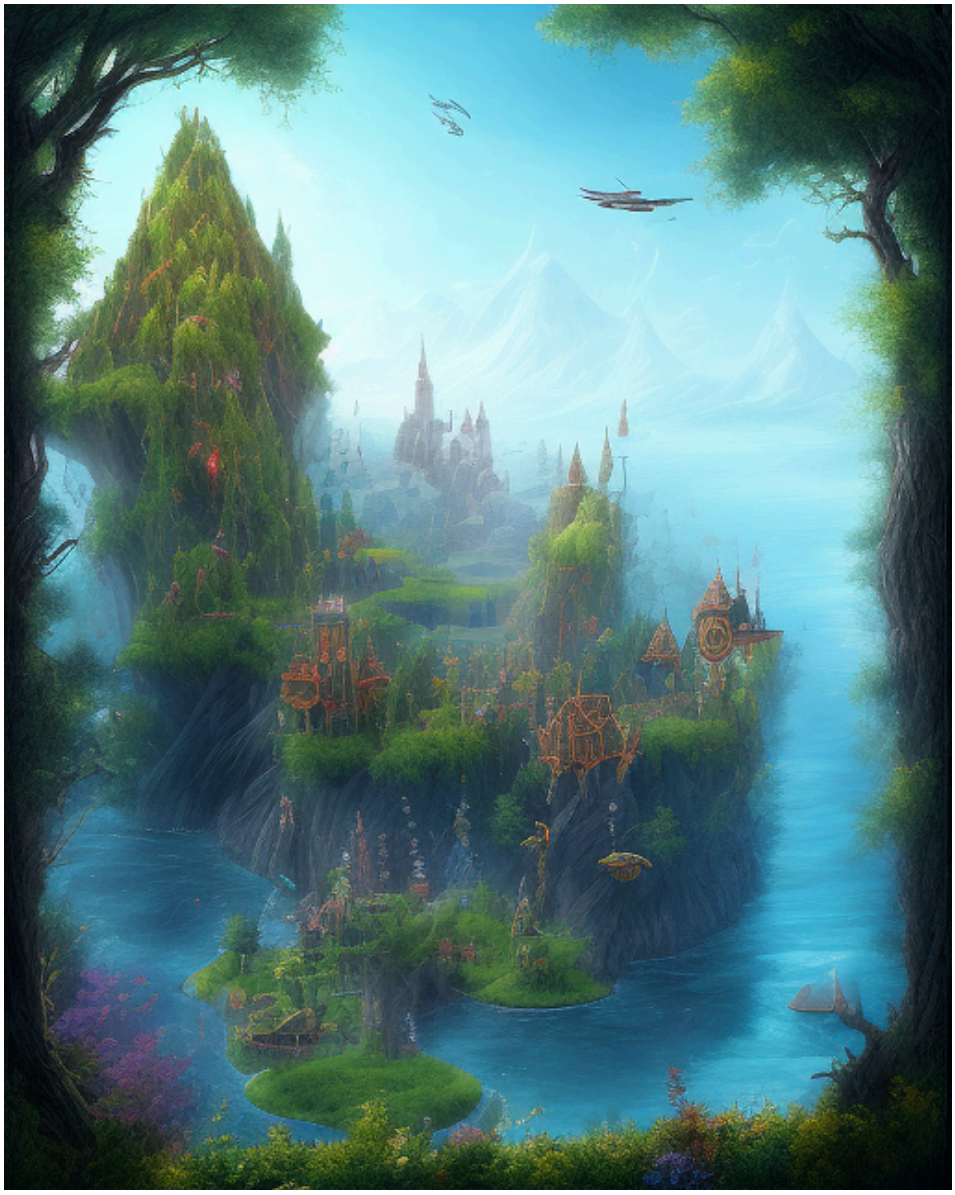
Input	**Output**	**Reason**
`"5", "6", "no dig"]`	`Too far`	`'5'` represents 5 as an integer `'6'` represents 6 as an integer Their sum is 11. 'no dig' has two words, the length of the first one is 2 and the length of the last one is 3. 2 + 3 is 5. Since 11 > 5, the function returns `'Too far'`

Hints:

- Casting will help here.
- What function should you reuse?

```
def digging(steps_left, steps_forward,  message):
    """
    >>> digging("5", "6", "no dig")
    'Too far'
    >>> digging("1", "4", "run away now!")
    'Too close'
    >>> digging("1", "8", "right here")
    'Dig here!'
    """
```

After digging, you discovered a picture of the Pythonia and more text on the back of the photo. The journey is about to begin!



Congratulations! You are done with Lab01 :)

Submission

You need to submit `lab01.py` via Gradescope. Follow [these steps] to submit your work.

Important Notes:

1. You may submit more than once before the deadline; only the final submission will be graded;
2. We will be grading this lab solely based on hidden tests, so you should test your code thoroughly by writing more test cases;

3. Unlike lab00, you should expect to see "-/10.0". You will see your score when the grades are released;
4. You should **wait for the Autograder to finish** running before leaving the site to ensure that the tests are properly run;
5. If your autograder fails to run, try to consult the Gradescope Common Errors section on the course website before making an ED post;
6. If you have any other questions, please post to ED.

Warning: If you used Virtual Studio Code as your editor, please double check if there are weird import statements that you don't recognize before submitting. Make sure to remove these as they will fail Gradescope runs!