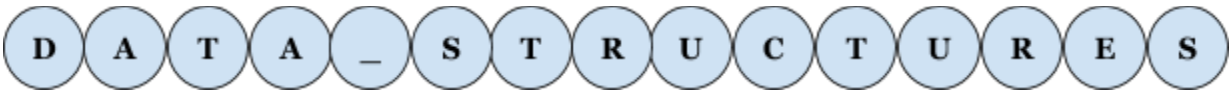
CS261 Data Structures

Assignment 1

v 1.03 (revised 6/22/2020)

Python Fundamentals Review



Contents

Assignment Summary 3

General Instruction 4

Specific Instructions 5

Program 1 - Divisibility 5
Program 2 - Min-Max 6
Program 3 - Matrix Addition 7
Program 4 - Swap Pairs 8
Program 5 - Camel Case 9

Summary

For this assignment, you will write a few short Python programs. There are several main objectives:

- Ensure that you are familiar with basic Python syntax constructs.
- Your programming environment is set up correctly.
- You are familiar with submitting assignments through Gradescope.

For this class, we assume you are comfortable with:

- Iterating over a list of elements using for or while loops
- Accessing elements in the list using index variable
- Passing functions as parameters to another functions
- Writing your own classes (including extending existing classes)
- Writing unit tests for your code
- Debugging your solutions

None of the programs in this assignment or in CS261 in general will require Python knowledge beyond what was covered in CS161 and CS162. If you completed CS161 / CS162 classes in Python, you should be able to complete this assignment. In case you need help, please post questions on Slack / Piazza or feel free to contact the instructors or the ULAs.

General Instructions

1. Programs in this assignment must be written in Python v3 and submitted to Gradescope before the due date specified in the syllabus. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

2. In Gradescope, your code will run through several tests. Any failed tests will provide a brief explanation of testing conditions to help you with troubleshooting. Your goal is to pass all tests.

3. We encourage you to create your own test programs and cases even though this work won't have to be submitted and won't be graded. Gradescope tests are limited in scope and may not cover all edge cases. Your submission must work on all valid inputs. We reserve the right to test your submission with more tests than Gradescope.

4. Your code must have an appropriate level of comments. At a minimum, each method should have a descriptive docstring. Additionally, put comments throughout the code to make it easy to follow and understand.

5. You will be provided with a starter "skeleton" code for each of the problems, on which you will build your implementation. Methods defined in skeleton code must retain their names and input / output parameters. Variables defined in skeleton code must also retain their names. We will only test your solution by making calls to methods defined in the skeleton code and by checking values of variables defined in the skeleton code. You can add more functions and variables, if you'd like, for your own use.

6. Both the skeleton code and code examples provided in this document are part of assignment requirements. Please read all of them very carefully. They have been carefully selected to demonstrate requirements for each method. Refer to them for the detailed description of expected method behavior, input / output parameters, and handling of edge cases.

7. For each method, you can choose to implement a recursive or iterative solution. When using a recursive solution, be aware of maximum recursion depths on large inputs. We will specify the maximum input size that your solution must handle.

8. If, after reading this entire document, you still have questions about any requirements of this assignment, please post your questions on Piazza. There is a specific thread for this assignment.

Problem 1 - Divisibility

Write a function that receives 4 integers (`m, n, a, b`) and for each number in the range [`m, n`] (inclusive) determines whether it is divisible by `a`, or it is divisible by `b`, or both `a` and `b` or by neither `a` nor `b` and returns the result as a list of strings, each string as separate list element. The numbers in the range must be processed in reverse order.

You may assume the input is always going to be positive integers. If any of the provided integers (`m, n, a,` or `b`) are non-positive, or if `m > n` or if `a` equals `b`, the program should just return a string (not a list) "Incorrect input".

Please refer to the examples below for the format of the output. Your program's output must match exactly. There is one TAB character (\t) between the columns.

Filename:	`a1_p1_divisibility.py`
Prototype:	`is_divisible(m: int, n: int, a: int, b: int) -> []`
Restrictions:	You can not use the built-in Python len() method. Write your own length() function that returns the length of the string you pass as argument to the function.

Example #1:

```
print(*is_divisible(2, 7, 2, 3), sep="\n")
```

Output:

```
Num Div by 2 and/or 3?
--- -------------------
7   None
6   both
5   None
4   div by 2
3   div by 3
2   div by 2
```

Example #2:

```
print(is_divisible(1, 20, -1, 3))
print(is_divisible(20, 0,  100, 200))
print(is_divisible(10, 8, 7, 2))
print(is_divisible(3, 30, 7, 7))
```

Output:

```
Incorrect input
Incorrect input
Incorrect input
Incorrect input
```

Problem 2 - Min-Max

Write a function that receives a one dimensional list of integers and returns a tuple with two values - minimum and maximum values in the input list. If the input list is empty, the function should return a tuple `(None, None)`.

Filename:	`a1_p2_minmax.py`
Prototype:	`min_max(arr: []) -> ()`
Restrictions:	You can not use built-in Python `min()` or `max()` methods on the input list. Your solution must iterate over elements in the list one-by-one. You can not use `sort()` or `sorted()` methods on the input array.

Example #1:

```
print(min_max([1, 2, 3, 4, 5]))
```

Output:

```
(1, 5)
```

Example #2:

```
print(min_max([8, 7, 6, -5, 4]))
```

Output:

```
(-5, 8)
```

Example #3:

```
print(min_max([]))
```

Output:

```
(None, None)
```

Problem 3 - Matrix Addition

Write a function that receives two two-dimensional matrices and returns the result of their addition. If matrices have different dimensions, the function should return None. You may assume the input is always going to be correct (all rows will have the same number of elements and all elements will be integers).

In case you need to refresh your knowledge of how to add matrices, check this wiki page: https://en.wikipedia.org/wiki/Matrix_addition

Filename:	`a1_p3_matrix_add.py`
Prototype:	`matrix_add(a: [[]], b: [[]]) -> [[]]`
Restrictions:	You can only use Python's standard library and are not allowed to import any other package(s). List elements can only be accessed by using an index variable. You are not allowed to use the `append()` method of Python lists.
Hints:	If you need to create a zero-filled matrix with M rows and N columns, one possible way of doing it is: `matrix = [[0] * N for _ in range(M)]`

Example #1:

```
m1 = [ [1, 2, 3], [2, 3, 4] ]
m2 = [ [5, 6, 7], [8, 9, 10] ]
m3 = [ [1, 2], [3, 4], [5, 6] ]

print(matrix_add( m1, m2 ))
print(matrix_add( m1, m3 ))
print(matrix_add( m1, m1 ))
print(matrix_add( [[]], [[]] ))
print(matrix_add( [[]], m1 ))
```

Output:

```
[[6, 8, 10], [10, 12, 14]]
None
[[2, 4, 6], [4, 6, 8]]
[[]]
None
```

Problem 4 - Swap Pairs

Write a function that receives a one dimensional list of integers and a returns a new list of the same length where pairs of elements are swapped (a[0] swapped with a[1], a[2] swapped with a[3] etc). Original list should not be changed. Finally, please remember that for a list of odd length, the last element shouldn't change.

Filename:	`a1_p4_swap_pairs.py`
Prototype:	`swap_pairs(arr: []) -> []`
Restrictions:	You can not use built-in list slices [::] functionality. Your solution must access list elements one at a time. For example, doing `arr[10]` is OK. Doing `a[0::2]` is not.
Hints:	One way to create a copy of the list is: `list_b = list(list_a)`

Example #1:

```
print(swap_pairs([1, 2, 3, 4, 5]))
```

Output:

```
[2, 1, 4, 3, 5]
```

Example #2:

```
print(swap_pairs([8, 7, 6, -5, 4, 10]))
```

Output:

```
[7, 8, -5, 6, 10, 4]
```

Example #3:

```
print(swap_pairs([]))
```

Output:

```
[]
```

Problem 5 - Camel Case

GENERAL DESCRIPTION

Write three functions which, working together, will convert a given input string to a "camel case" string using specific rules described below. Input string will consist of any printable ASCII characters (english letters, digits, spaces, underscores, some special characters). Examples of possible input strings are: `"_random_ _word_provided"`, `"@$ptr*4con_"`, `" ran  dom  word"`, `"_ _ random_"`.

Valid input string will consist of at least two groups of alphabetic characters, not necessarily forming actual words in the dictionary, separated by any non-alphabetic characters. All valid strings will be converted to "camel case". All invalid strings will not be processed.

Filename:	`a1_p5_camel_case.py`

Prototypes:	`input_cleanup(input_string: str) -> str`
	`is_clean_string(input_string: str) -> bool`
	`camel_case(s: str, func_is_clean, func_cleanup) -> str`

Restrictions:	The following built-in Python functions are not allowed for this problem: 1) Any string processing methods (lower(), upper(), capitalize(), find(), index() etc.). Here is a fairly comprehensive list of methods that are NOT allowed: https://www.w3schools.com/python/python_ref_string.asp 2) len() method (please use `for letter in input_str` construct to iterate over a string letter-by-letter). You're also welcome to write your own implementation of len() method using prototype provided in the skeleton code.
Hints:	To convert between lowercase and uppercase letters, you can use `ord()` and `chr()` built-in methods. You're also welcome to use slices for this problem, like: `str[start: stop: step]`

Please see the following pages for detailed specifications and usage examples for each function.

DETAILED SPECS - input_cleanup()

```
input_cleanup(input_string: str) -> str
```

This function receives the original input string from the user and returns a 'clean' version of it. 'Clean' is defined as follows:

1) All english letters (a-z A-Z) have been converted to lowercase.
2) All characters other than english letters are treated as word separators.
3) All word separators are placed by underscore '_' characters.
4) All duplicate word separators have been removed.
5) All leading (occurring before any text) and trailing (occurring after all text) word separators have been removed.

Example #1:

```
test_set = ("_random_ _word_provided",
          "@$ptr*4con_", " ran  dom  word",
          "example    word   ", "ANOTHER_Word",
          "__", "_ _ _", "    ", "435%7_$$", "random")

for test_string in test_set:
    result = input_cleanup(test_string)
    print(length(result), result)
```

Output:

```
20 random_word_provided
7 ptr_con
12 ran_dom_word
12 example_word
12 another_word
0
0
0
0
6 random
```

DETAILED SPECS - is_clean_string()

`is_clean_string(input_string: str) -> bool`

This function verifies whether input string is 'clean' and is ready for a 'camel case' conversion according to the rules below:

1) Input string contains only lowercase english letters (a-z) and underscores (_).
2) Does not contain any duplicate underscore characters (like __).
3) Does not contain any leading or trailing underscores (like "_test_case" or "test_case_").

If ALL the requirements above are satisfied, the function will return True. Otherwise, it should return False.

Note that this function is essentially a check whether `input_cleanup()` has properly done its job. If you pass any output from `input_cleanup()` into `is_clean_string()`, the second function should always return True.

Example #2:

```
test_set = ("_random_ _word_provided",
         "@$ptr*4con_", " ran  dom  word",
         "example    word   ", "ANOTHER_Word",
         "__", "_ _ _", "    ", "435%7_$$", "random")

for test_string in test_set:
    result = input_cleanup(test_string)
    print(is_clean_string(test_string), is_clean_string(result))
```

Output:

```
False True
False True
False True
False True
False True
False True
False True
False True
False True
True True
```

DETAILED SPECS - camel_case()

`camel_case(input_string: str, func_is_clean, func_cleanup) -> str`

This function accepts the original input string from the user, as well as functions `is_clean_string()` and `input_clean()` that you have implemented above.

It then 'cleans up' the original user string by calling `func_cleanup()` on it and verifies that the resulting string is in fact 'clean' by calling `func_is_clean()` function on it. This part has already been pre-written for you in the skeleton file and should not be changed.

After the input string has been 'cleaned up', `camel_case()` will process it by:

1) Capitalizing all letters that immediately follow underscore characters.
2) Removing word separators (underscore characters) from the output string.

This function should not process the input string and just return None if either of the following conditions is true:

1) `func_is_clean()` returned False for some reason
2) Input string does not have at least two words in it (separated by underscores).

In all other instances, it should return a 'camel case' version of the 'cleaned up' input string.

Example #3:

```
test_set = ("_random_ _word_provided",
            "@$ptr*4con_", " ran  dom  word",
            "example    word   ", "ANOTHER_Word",
            "__", "_ _ _", "    ", "435%7_$$", "random")

for test_string in test_set:
    result = camel_case(test_string, is_clean_string, input_cleanup)
    print("'" + test_string + "'", "-->", result)
```

Output:

```
'_random_ _word_provided' --> randomWordProvided
'@$ptr*4con_' --> ptrCon
' ran  dom  word' --> ranDomWord
'example    word   ' --> exampleWord
'ANOTHER_Word' --> anotherWord
'__' --> None
'_ _ _' --> None
'    ' --> None
'435%7_$$' --> None
'random' --> None
```