ECS 32A: Programming Assignment #4

Instructor: Aaron Kaloti

Summer Session #1 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Added clarification to part #6.
- v.3: Autograder details. Clarification to part #7.

2 Due Date

This assignment is due the night of Friday, July 17. Gradescope will say 12:30 AM on Saturday, July 18, due to the "grace period" (as described in the syllabus). *Do not rely on the grace period for extra time; this is risky.*

Some students tend to email me very close to the deadline. This is also a bad idea. There is no guarantee that I will check my email right before the deadline.

3 Manual Review

See the style guide on Canvas here. A small portion of your grade for this assignment will be decided by me or one of the TAs; one of us will manually check your code and check that you made a decent effort at following the study guide. Blatant violations that suggest you didn't even look at the study guide could very easily earn you a zero for the manual review portion of your score.

Do not ask (over email or office hours) questions such as, "Could you take a look at my code and tell me if I will get a 100% for the manual review portion?" You should, however, ask about specific components of the style guide that you are not sure about.

4 Grading Breakdown

This assignment is worth 9% of your final grade and will be worth 90 points in the autograder. Here is the breakdown of the 90 points by part:

- Part #1: 10
- Part #2: 10
- Part #3: 10
- Part #4: 15 (5 for `max_length()` and 10 for `interleave()`)
- Part #5: 10
- Part #6: 10
- Part #7: 10
- Part #8: 10
- Manual review (see above section): 5

5 Problems

All code should be placed in a file called `prog4.py`.
You are not allowed to `import` any modules.

5.1 Part #1: ESS and DESS

One computational theory (ECS 120) textbook defines the ESS ("Encode as Single String") encoding of two strings as a single string that begins with the length of the first string, followed by a space, then followed by the first string itself, and finally followed by the second string. Write two functions:

1. a function called `ess` that takes as arguments two strings and returns the ESS encoding of the two strings.
2. a function called `dess` (for "DEcode from Single String") that takes as its two arguments a string that is *assumed* to be in ESS encoding and an integer that must be 1 or 2. The function should then return the first or second string (depending on the second argument) that is encoded in the first argument. Although you may assume the second argument is always an integer, your function should return "ERROR" if the second argument is not either 1 or 2. You may assume that the length of the first string within the encoding will never exceed 9.

Below are examples of how your functions should behave. As a reminder, `ess()` and `dess()` return strings. When you print strings by using `print()`, you don't see single quotes around them; when you show the strings without using `print()` on the interpreter, the single quotes are shown around them. That's why you see single quotes around the strings shown below when `print()` is not used.

```
>>> ess("abc","hi")
'3 abchi'
>>> print(ess("abc","hi"))
3 abchi
>>> ess("xxx","xxxx")
'3 xxxxxxx'
>>> ess("hello","world")
'5 helloworld'
>>> ess("$#!^","@#%")
'4 $#!^@#%'
>>> dess('3 abchi',1)
'abc'
>>> print(dess('3 abchi',1))
abc
```

```
>>> dess('3 abchi',2)
'hi'
>>> dess('3 xxxxxxx',2)
'xxxx'
>>> dess('5 helloworld',1)
'hello'
>>> dess('5 helloworld',2)
'world'
>>> dess('4 $#!^@#%',1)
'$#!^'
>>> dess(ess("xd$x","$%"),1)
'xd$x'
>>> dess(ess("xd$x","$%"),2)
'$%'
```

5.2 Part #2: Echo

Write a function called `echo` that takes as argument a string and returns a copy of that string with each character duplicated right after it. Below are examples of how your function should behave.

```
>>> echo("abc")
'aabbcc'
>>> print(echo("abc"))
aabbcc
>>> echo("xy")
'xxyy'
>>> echo("hi there")
'hhii  tthheerree'
```

5.3 Part #3: Find Last Mismatch

Write a function called `find_last_mismatch` that takes as arguments two strings and returns the last (positive) index at which the two strings differ. If the strings are identical, then the function should return -1. You may assume that the strings have the same length.

Below are examples of how your function should behave.

```
>>> find_last_mismatch("abcd", "axyz")
3
>>> find_last_mismatch("abc", "abd")
2
>>> find_last_mismatch("abc", "a c")
1
>>> find_last_mismatch("abc", "abc")
-1
>>> find_last_mismatch("", "")
-1
```

5.4 Part #4: Interleave

First, write a function called `max_length` that takes as argument three strings and returns the *length* of the largest of the three strings. **You may not use the built-in `max()` to implement this function; you must use conditional statements.**

Next, write a function called `interleave` that takes as argument three strings and returns a string that is the result of interleaving the given strings together, one character at a time. For example, if the three arguments are "abc", "ABC", and "xyz", then the return value should be "aAxbBycCz". If the three strings are of unequal lengths, then the shorter strings should be treated as if they have enough trailing whitespaces to make the strings be of equal length. For example, if the three arguments are "ab", "xyz", and "X", then the return value should be "axXby z ". In this example, "ab" is treated like "ab " (1 trailing space), and "X" is treated like "X " (2 trailing spaces), in order to match the length of "xyz". **You must use `max_length()` in this function.**

Hint (or *Challenge*?): If you set things up properly before your loop, you can implement `interleave()` without using conditional statements. My version with conditional statements has 16 lines in its body; my version *without* conditional statements has 8 lines in its body.

Below are some examples of how your functions should behave.

```
>>> max_length("abc","de","xxyy")
4
>>> max_length("abc","de","xx")
3
>>> max_length("abCCC","","xyx")
5
>>> max_length("","","x")
1
>>> max_length("","","")
0
>>> interleave("xxx","yyy","zzz")
'xyzxyzxyz'
>>> print(interleave("xxx","yyy","zzz"))
xyzxyzxyz
>>> interleave("xxx","yyy","zz")
'xyzxyzxy '
>>> interleave("xx","yyy","zz")
'xyzxyz y '
>>> interleave("xx","yy","zz")
'xyzxyz'
>>> interleave("x x","yy","zz")
'xyz yzx  '
>>> interleave("","abc","def")
' ad be cf'
>>> interleave("","abc","de")
' ad be c '
```

5.5 Part #5: Get Longest Stretch

Write a function called `get_longest_stretch` that takes a string and a target character as argument and returns the length of the longest stretch of the target character in the given string.

Hint: Although it may not seem like it at first, this problem is doable with one loop.

Below are examples of how your function should behave.

```
>>> get_longest_stretch("abcd", "a")
1
>>> get_longest_stretch("abcd", "e")
0
>>> get_longest_stretch("abccd", "c")
2
>>> get_longest_stretch("Hello there", "l")
2
>>> get_longest_stretch("xyyzyxyyyz", "y")
3
>>> get_longest_stretch("effghggefhhhg", "g")
2
>>> get_longest_stretch("effghggefhhhg", "f")
2
>>> get_longest_stretch("effghggefhhhg", "h")
3
>>> get_longest_stretch("effghggefhhhg", "e")
1
>>> get_longest_stretch("effghggefhhhg", "a")
0
```

5.6 Part #6: Compute Product At

Write a function called `compute_product_at` that takes as argument two lists of integers. The second list is a list of indices, all unique. `compute_product_at()` should return the product of the values in the first list that are at the indices given in the second list. This is easier to explain with examples, which are shown below. None of the indices in the second list will be out-of-range.

You may assume that neither list will ever be empty.

In this first example, the returned value is `1200`, because that is the product of the integers in the first list that are at indices 1, 5, and 0. (That is, if we refer to the first list as `vals`, then the returned value comes from the fact that `vals[1] * vals[5] * vals[0]` $= 15 \cdot 10 \cdot 8 = 1200$.)

```
>>> compute_product_at([8,15,3,-2,1,10],[1,5,0])
1200
```

In this second example, the returned value is `-6`, because that is the product of the integers in the first list that are at indices 3, 4, and 2.

```
>>> compute_product_at([8,15,3,-2,1,10],[3,4,2])
-6
```

5.7 Part #7

Write a function called `insert_name_here`[1] that takes as argument a list of integers (we'll call this list `vals`) and returns a list containing each integer in `vals` that is greater than the integers immediately before and after it.

In the case of the first element of `vals`, it only needs to be greater than the element after it to be included. In the case of the last element of `vals`, it only needs to be greater than the element *before* it to be included. If `vals` is empty or only contains one element, then `insert_name_here()` should return an empty list.

The values in the returned list must maintain the order that they had in the list that was passed in. For example, in the second case shown below (`insert_name_here([-1,1,0,5])`), it would be wrong to return the list `[5, 1]` instead of `[1, 5]`.

Below are examples of how your function should behave.

```
>>> insert_name_here([2,5,8,3])
[8]
>>> insert_name_here([-1,1,0,5])
[1, 5]
>>> insert_name_here([3,2,2,4])
[3, 4]
>>> insert_name_here([3,2,2,3])
[3, 3]
>>> insert_name_here([10,8,-3,70,15,22,20,19,-5])
[10, 70, 22]
>>> insert_name_here([200,100,300,450,570,100,120])
[200, 570, 120]
```

5.8 Part #8

Write a function called `get_key_to_min` that takes a dictionary and returns the key of the smallest value in the dictionary that is greater than or equal to 10.

About the values in the given dictionary, you may assume:

1. They are all integers that do not exceed 1000.
2. They are all unique (i.e. different from each other).
3. At least one of the values is at least 10.

Here are some examples of how your program should behave.

```
>>> get_key_to_min({'a':5,'b':8,'abc':13,'def':18,'xyz':10})
'xyz'
>>> get_key_to_min({'a':5,'b':8,'abc':13,'def':18,'xyz':9})
'abc'
>>> get_key_to_min({'a':12,'b':8,'abc':13,'def':18,'xyz':9})
'a'
```

6 Autograder Details

All of the details about the autograder that were given in the directions for the previous programming assignment are still relevant here, so I do not repeat them here.

6.1 Test Cases' Inputs

Except where otherwise stated, each part has five cases (three visible, two hidden).

Be careful about copy/pasting from PDFs, particular certain marks (single quotation marks and maybe double quotation marks); certain characters tend to get distorted when copy/pasted from PDFs.

[1] As a student pointed out to me, `find_local_maxima` would have been a better name.

6.1.1 Part #1

This part has ten cases: five for `ess()` and five for `dess()`. For each of these five, the last two are hidden.
Case #1:

```
ess("squidward", "smells")
```

Case #2:

```
ess("abc de", "ab cde")
```

Case #3:

```
ess("", "xyz")
```

Case #6:

```
dess("1 abcdefghijk", 1)
```

Case #7:

```
dess("1 abcdefghijk", 2)
```

Case #8:

```
dess("5 aabbaabb", 3)
```

6.1.2 Part #2

Case #1:

```
echo("apple")
```

Case #2:

```
echo("xYzXyZ")
```

Case #3:

```
echo("a b c d")
```

6.1.3 Part #3

Case #1:

```
find_last_mismatch("apple", "appXe")
```

Case #2:

```
find_last_mismatch("abcdeef", "a_c_e_f")
```

Case #3:

```
find_last_mismatch("abcdefghijk", "abcdefghijk")
```

6.1.4 Part #4

For all of the test cases below, an underscore (_) denotes a whitespace, NOT an actual underscore. For example, "xx_" corresponds to "xx ". The first three cases are for `max_length()`; the others are for `interleave()`.
Case #1:

```
aaaa
bbbb
cccc
```

Case #2:

```
xxx
xxx
xxx
```

Case #3:

```
xx_
x_x
_xx
```

Case #4:

```
abc

def
```

Case #5:

```
a__b
_cd_
f_g
```

Case #6:

```
abb

bba
```

Case #7:

```
abcde
fgh
ijkl
```

Case #8:

```

abc
defg
```

6.1.5 Part #5

Case #1:

```
get_longest_stretch("xyzzyzyz", "c")
```

Case #2:

```
get_longest_stretch("effghggefhhhg", "h")
```

Case #3:

```
get_longest_stretch("aXXbXcXddXXXeeeeeeeeXXXXfXg", "X")
```

6.1.6 Part #6

Case #1:

```
compute_product_at([8,15,3,-2,1,10],[3,2,4])
```

Case #2:

```
compute_product_at([7,100,2,30,4,20],[5,0])
```

Case #3:

```
compute_product_at([9,8,7,6,5,4,3,2,1,0],[1,3,7,6])
```

6.1.7 Part #7

Case #1:

```
insert_name_here([2,8,5,3])
```

Case #2:

```
insert_name_here([18,7,-5,20,3,-30,14,7])
```

Case #3:

```
insert_name_here([7])
```

6.1.8 Part #8

Case #1:

```
get_key_to_min({'200': 20, '10': 1, '300': 30, '250': 25})
```

Case #2:

```
get_key_to_min({83: 2, 55: 79, 61: 11, 62: 9})
```

Case #3:

```
get_key_to_min({'abc': 11,'def':10,'efg':12})
```