Homework 3 Instructions

UCSD Extension CSE-41273, Summer 2022.

The zipped file contains 2 files (besides this one), `HW3.py` and `HW3_cli.py`.

Please review the *About Homework* document for reminders and pointers. When you are finished with the homework, compress the two files **together** (not a folder, just the 2 files) into a zip file named `HW3.zip` and upload it to Canvas.

Read the instructions VERY carefully! Read **everything** before you start coding. If you have any questions please email me! There is no test program provided. You need to do your own testing!

Part 1. The file `HW3.py` is a module file with function skeletons in it, just like we've seen before. There is no separate test program; you will have to use the guidelines from the instructions to determine if your code is working correctly. Each function has documentation (below) about what it is supposed to do and how you can run the functions in the REPL to do manual testing. If you think you need to `import` any libraries, please contact me so I can tell you why you shouldn't `import` anything.

Part 2. The second file, `HW3_cli.py`, is a skeleton of a CLI program. The program will accept one or more input strings and up to 3 flags to determine what to do with the input strings. The details are described below.

Part 1 - Details of the functions in 'HW3.py'

Nothing in `HW3.py` should print anything!

len_safe (10 points)

Edit the function `len_safe` such that it does the following. It **returns** (does not print) the length of the input object as given by the function `len()`, or it returns `-1` (the **number** `-1`, not a string) if the object has no length. Use exception handling for this one.

What does it mean for an object to have no length? It is an object for which the `len()` method does not apply. What happens when you try to get the `len` of something that has no `len`? Try doing `len(5)` in the REPL. What happens? We want our len_safe function to return `-1` for this situation, instead of getting an error. We need to capture that specific exception and return `-1` for objects that have no `len`. Remember, you should only capture (in an `except`) the errors that *you know how to deal with*.

It should work like this in the REPL:

```
>>> from HW3 import len_safe # Or, copy/paste the function.
>>> my_dict = {'a': 23, 'b': 8}
>>> len_safe(my_dict)
2
>>> len_safe([]) == 0
True
>>> len_safe(0.25) == -1
True
>>> len_safe(7)
-1
>>> len_safe(None)
-1
>>> len_safe('cat')
```



```
3
>>> len_safe('')
0
>>> animals = ['dog', 'cat', 'bird', 'cat', 'fish']
>>> len_safe(animals)
5
>>> import math
>>> len_safe(math.pi)
-1
```

list_pairs (10 points)

Edit the function `list_pairs` such that, given a list, it returns a list of tuples. For each tuple, the first element is an item from the list, and the second element of the tuple is the following item of the list. One of the built-in functions that was introduced in the lecture on looping should be helpful. Also you might want to consider how slicing could be useful as part of this problem.

You must use a single list comprehension. There should be no other code outside the list comprehension.

It should work like this:

```
>>> from HW3 import list_pairs # Or, copy/paste the function.
>>> numbers = [1, 2, 3, 5, 7, 8]
>>> result = list_pairs(numbers)
>>> result
[(1, 2), (2, 3), (3, 5), (5, 7), (7, 8)]
>>> list_pairs([])
[]
>>> list_pairs([1])
[]
>>> list_pairs([1, 2])
[(1, 2)]
>>> animals = "cats, dogs, horses, hamsters, birds".split()
>>> list_pairs(animals)
[('cats,', 'dogs,'), ('dogs,', 'horses,'), ('horses,', 'hamsters,'), ('hamsters,', 'birds')]
```

unique (20 points)

Edit the function `unique` to create a *generator* function that loops over the input iterable sequence, yielding one element at a time, but skipping duplicates. Note the output should be in the *same order* as the input sequence. You may assume that all elements in the input iterable are atomic, namely they are all single objects and not lists or dictionaries, etc. **It must not modify the input sequence object.**

Note that it should be a *generator function*. This means that in the `unique` function, **do not make a new list containing all the unique elements** and *then* produce the output. **Do not use any methods of `dict` or import any libraries.** However, you may want to use a set or list as a helper. For best results, use `yield`.

It should work like this in the REPL:

```
>>> from HW3 import unique # Or, copy/paste the function.
>>> numbers = [4, 5, 2, 6, 2, 3, 5, 8]
>>> nums = unique(numbers)
>>> next(nums)
4
>>> next(nums)
5
>>> next(nums)
2
>>> next(nums)
```

```
6
>>> next(nums)
3
>>> next(nums)
8
>>> next(nums)
Traceback (most recent call last):
  [...]
StopIteration
>>>
>>> things = unique(['dog', 'cat', 'bird', 'cat', 'fish'])
>>> next(things)
'dog'
>>> next(things)
'cat'
>>> next(things)
'bird'
>>> next(things)
'fish'
>>>
>>> next(things)
Traceback (most recent call last):
  [...]
StopIteration
```

Note: I put the `[...]` to represent the traceback from Python. Your traceback might be slightly different from mine, so I use that as a placeholder.

When you are testing your code, it is possible to test whether you have the correct *values and order* by putting the generator returned from `unique` into a list using the list constructor function: `list(unique(values))`. This tells you if it returns the correct values, but it does **not** tell you if it is an iterator/generator. To be sure your code works, you need to make sure that you can call `next()` on the object you get back from `unique()` and that you get a `StopIteration` error when all the correct elements have been returned with `next()`.

Part 2 Details of writing 'HW3_cli.py'

Use `argparse` to implement the CLI portion of the program so it works as shown here. It is one of the few programs in this course that actually prints output.

Output from the program should look **exactly** like this when you use the `-h` help flag. **Hint:** You get the `-h` help flag for free with argparse. You do not have to implement the `-h` flag yourself.

```
$ python HW3_cli.py -h
usage: HW3_cli.py [-h] [-p] [-c] [-l] texts [texts ...]

positional arguments:
  texts          Input strings to operate on

optional arguments:
  -h, --help     show this help message and exit
  -p, --print    Just print the input strings
  -c, --combine  Print input strings combined in a continuous string
  -l, --len      Print each string with its index and length

Does print, combine, then len. If no flags given, does nothing.
```

See that description line at the end telling what it does? It is created with the epilog.

NOTE: I show `optional arguments:` in the above example of the help flag. In some versions of Python, that will be `options:`. Do not worry! This is OK, my testing takes that into account. Everything else should be exactly as shown.

If no arguments are given at all, it should give an error that the `texts` arguments are required. **Hint:** You get this error for free if you program the `texts` argument correctly.

```
$ python HW3_cli.py
usage: HW3_cli.py [-h] [-p] [-c] [-l] texts [texts ...]
HW3_cli.py: error: the following arguments are required: texts
```

The behavior of the three flags are explained below. If at least one input string is given, but no flags are given, the program should **do nothing**. Since there are no flags used in the lecture examples, you will need to check the `argparse` documentation (look at the tutorial here) to find how to implement flag arguments. They are called "Short options" in the documentation. Look for the example of implementing the `-v` "verbose" option.

Arguments

The flag arguments control what the program does. The flags can be given in any order or combination on the command line, however, these are the rules for implementation: **They should be implemented in the order shown below** - in other words, the `print` flag (if given) is executed first, then the `combine` flag (if given), then the `len` flag (if given). **Please make all the help strings exactly as shown above in the output from using help, as shown above.** Feel free to copy/paste from this document.

The `-p` or `--print` flag will print out the input strings with spaces in between each string. We know how to make one string from a list of strings with one of the str methods.

The `-c` or `--combine` flag will print all the input strings concatenated together. Again, we know how to do that with a string method.

The `-l` or `--len` flag prints out a message for each string with the index, the string itself, and the length of the string.

No, you don't need to use `len_safe`, since they are always strings. The format of the message is shown below in the examples.

The program collects the input strings from the command line as positional arguments into the argparse variable `texts`. At least one input string is required. You will need to adjust `nargs` for `texts` to allow one or more input strings - ie, one or more positional arguments are required. See the docs for nargs for details.

Examples of how it should work. Note that I added a blank line between some commands for clarity:

```
$ python HW3_cli.py -p These Strings Get Printed
These Strings Get Printed

$ python HW3_cli.py -p -l These Strings Get Printed
These Strings Get Printed
Word #1 is "These" and has 5 letter(s)
Word #2 is "Strings" and has 7 letter(s)
Word #3 is "Get" and has 3 letter(s)
Word #4 is "Printed" and has 7 letter(s)

$ python HW3_cli.py -c These Strings Get Concatenated
TheseStringsGetConcatenated

$ python HW3_cli.py -c -l These Strings Get Concatenated
TheseStringsGetConcatenated
Word #1 is "These" and has 5 letter(s)
Word #2 is "Strings" and has 7 letter(s)
Word #3 is "Get" and has 3 letter(s)
Word #4 is "Concatenated" and has 12 letter(s)

$ python HW3_cli.py -c -p These Strings Get Printed And Concatenated
These Strings Get Printed And Concatenated
TheseStringsGetPrintedAndConcatenated

$ python HW3_cli.py -l -c -p These Strings Get Printed And Concatenated
These Strings Get Printed And Concatenated
TheseStringsGetPrintedAndConcatenated
Word #1 is "These" and has 5 letter(s)
Word #2 is "Strings" and has 7 letter(s)
Word #3 is "Get" and has 3 letter(s)
Word #4 is "Printed" and has 7 letter(s)
Word #5 is "And" and has 3 letter(s)
Word #6 is "Concatenated" and has 12 letter(s)

$ python HW3_cli.py --combine --len --print These Strings Get Printed And Concatenated
These Strings Get Printed And Concatenated
TheseStringsGetPrintedAndConcatenated
Word #1 is "These" and has 5 letter(s)
Word #2 is "Strings" and has 7 letter(s)
Word #3 is "Get" and has 3 letter(s)
Word #4 is "Printed" and has 7 letter(s)
Word #5 is "And" and has 3 letter(s)
Word #6 is "Concatenated" and has 12 letter(s)

$ $ python HW3_cli.py --len --combine a b c d e
abcde
Word #1 is "a" and has 1 letter(s)
Word #2 is "b" and has 1 letter(s)
Word #3 is "c" and has 1 letter(s)
Word #4 is "d" and has 1 letter(s)
Word #5 is "e" and has 1 letter(s)

$ python HW3_cli.py -l testing
Word #1 is "testing" and has 7 letter(s)
$ python HW3_cli.py -p testing
testing
$ python HW3_cli.py -c testing
testing
```

```
$ python HW3_cli.py  testing
```

Note the last one has a string, but has no flags to tell the program what to do, so it does *nothing*.

Also, if you get things showing up in **SHOUTY CAPITALS** in the *help output*, then you need to follow the link to the Argparse tutorial to find how to code a *short option*.

Grading 100 points total

`HW3.py` totals 40 points. `HW3_cli.py` totals 60 points: Each argument setup is 8 points (32) points). Implementing each flag in order is: 4 points each for `--print` and `--combine`; 10 points for `--len`. The Epilog is 4 points.

My email is dianechen.ucsdext@gmail.com. Please do not hesitate to email me if you have questions.

