ECS 32B: Programming Assignment #1

Instructor: Aaron Kaloti

Summer Session #2 2020

Contents

1 Changelog

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

- v.1: Initial version.
- v.2: Fixed one of the `ValueError` example messages in part #3.
- v.3: Added and clarified conditions for invalid input in part #4.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Thursday, August 13. Gradescope will say 12:30 AM on Friday, August 14, due to the "grace period" (as described in the syllabus). *Rely on the grace period for extra time at your own risk.*

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 Grading Breakdown

Each part will probably contribute 25% to this assignment's total worth, but the exact breakdown will be revealed once the autograder is released.

4 Autograder Details

Once the autograder is released, I will update this document to include important details here. Note that you should be able to verify the correctness of your functions without depending on the autograder. In order to incentivize you to become used to this, I tend not to release the autograder until around two or three days before the deadline.

4.1 Reminder about Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission, which I will talk about at the beginning of the Tuesday (8/11) lecture.

5 Problems

All of your functions should be placed in a file called `prog1.py`. You are not allowed to have any other files in your submission. You are of course allowed to write helper functions.

Restrictions:

- **Permitted modules**: You are not allowed to `import` any modules, except for the following modules:
  - `person` (see part #4)
  - `copy`
  - `typing`
    - This module helps with "type annotations", which let you specify what types you expect function arguments or return values to have. As stated in the Python documentation here, "The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as type checkers, IDEs, linters, etc." I do not require or encourage[1] that you use type annotations, but you may find them worth trying out.

```
from typing import List
...
def find_matches(string: str, target: str) -> List[int]:
    pass
```

- **Permitted methods**: Of all of the string, list, and dictionary methods, only the below ones are permitted[2]. Any method not listed below is prohibited. As you may have learned in ECS 32A, a method is a special kind of function that is called by specifying a value (usually a variable) followed by a dot/period and finally followed by the function call. For example, `split()` is a method and can be called through something like `s.split()`, where `s` is a string. Conversely, `len()` is not a method, because you would call `len()` by doing `len(s)` instead of `s.len()`, so the restrictions specified here have nothing to do with `len()`.
  - String methods: `format()`, `split()`.
  - List methods: `append()`, `extend()`.
  - Dictionary methods: `items()`, `values()`, `setdefault()`, `get()`.

5.1 Part #1

Write a function called `remove_vals` that takes two arguments: a list of lists of integers (that we'll call `vals` here) and a list of integers (that we'll call `targets`). `remove_vals()` should return a list of lists of integers; this list should be identical to `vals`, except with any value that appears in `targets` removed. You are not allowed to modify `vals`, and you must create a new list of lists of integers.

Hint: If you do not properly use the `in` operator, you may find that you need triple nested loops.

Below are some examples of how your function should behave on a Python interpreter.

```
>>> remove_vals([[20, 13, 17, 19, 4], [5, 6], [8, 22, 3], [13, 12], [42, 58, 97, 30]], [13, 6, 97, 50, 42,
    -1])
[[20, 17, 19, 4], [5], [8, 22, 3], [12], [58, 30]]
>>> remove_vals([[20, 13, 17, 19, 4], [5, 6], [8, 22, 3], [13, 12], [42, 58, 97, 30]], [4, 5, 6, 7, 8])
[[20, 13, 17, 19], [], [22, 3], [13, 12], [42, 58, 97, 30]]
```

[1] Another instructor has his students use this in his ECS 32B course, but I do not think that I agree with this decision. For one thing, I believe that one of the main reasons for which one would prefer Python over C++ is that they do not have to explicitly specify the type of all variables and return values in Python.

[2] In the interest of transparency, I will say that the reason for this restriction is that I do not want you to use certain methods that make certain parts too easy. In the real world, you would obviously use all of the methods available to you so as to avoid writing code that does the same thing as code that is already available, but ECS 32B is not the real world.

5.2 Part #2

Write a function called `find_matches` that takes two strings as arguments. `find_matches()` should return a list of integers containing all nonnegative indices (in ascending order) at which the second argument occurs in the first argument. Put informally, `find_matches()` determines all of the locations at which the second string appears in the first string.

Below are some examples of how your function should behave on a Python interpreter.

```
>>> find_matches("abcdefghijk", "f")
[5]
>>> find_matches("abcdefghijk", "x")
[]
>>> find_matches("abracadabra", "br")
[1, 8]
>>> find_matches("aabbcdeebbcbcdbcde", "bcd")
[3, 11, 14]
```

5.3 Part #3

Write a function called `update_sales` that takes two arguments:

- `curr_sales`: A dictionary that maps strings to integers (i.e. a dictionary in which the keys are strings and the values are integers). This dictionary represents the current sales of items.
- `filename`: The name of a file (see below).

Each line in the named file should contain an item's name followed by the amount of that item sold recently. In the below example, the file says that four helmets, two wallets, and one keychain were sold. `update_sales()` should *modify* `curr_sales` based on the changes specified in the file. For example, if the dictionary were {`'wallet': 4, 'phone': 8, 'gum': 20, 'keychain': 2`}, then after `update_sales()` is called[3], the dictionary should have been modifed so that it is {`'wallet': 6, 'phone': 8, 'gum': 20, 'keychain': 3, 'helmet': 4`}. You can see that the values corresponding to `"wallet"` and `"keychain"` were increased by 2 and 1, respectively, and that since `"helmet"` was not a key in the original dictionary, it was given a new key-value pair.

```
helmet 4
wallet 2
keychain 1
```

`update_sales()` must `raise` a `ValueError` in any of the following cases. In such cases, you need not worry about undoing any changes that your function may have made to the dictionary before raising the `ValueError`. (The message associated with your `ValueError` will not be checked.)

- A line (in the file) in which the item name or the sales is missing is encountered.
- A line with too much information (i.e. more than just the item name, which must be one word, and the sales, which must be an integer) is encountered.
- An item's sales is not a positive number.

Your function must close any file that it opens. You may assume that each line of the text file ends in a newline character[4].

Below are the contents of a few example text files and interactions with `update_sales()` on the Python interpreter.

`infile1.txt`:

```
helmet 4
wallet 2
keychain 1
```

`infile2.txt`:

```
gum 13
helmet 2
paper 1
```

`infile3.txt`:

```
notebook 2
bottle -2
```

[3] By the way, be careful about copying strings from this document. Whenever one copies single quotation marks (and maybe double quotation marks as well) from a PDF that styles such marks in a certain way (like LaTeX-based PDFs do), the quotation mark will be distorted when it is pasted into your Python code or interpreter, and Python will not recognize it as the mark that it is supposed to be.

[4] The reason I say this is that when you are editing a text file with certain editors (e.g. Sublime Text, I think), the text file will not be automatically given a newline character at the end of the *last line*, and this can cause some unexpected issues in your function that I do not want you to worry about. However, I believe that the "official" definition of a "text file" (as opposed to a "binary file"), set by the POSIX standard is that each line of a text file ends with a newline character. (Note that here, a "text file" does not imply a file whose file extension is `.txt`.)

infile4.txt:

```
notebook 2
toilet paper 10
```

Interpreter interaction:

```
>>> d = {'wallet': 4, 'phone': 8, 'gum': 20, 'keychain': 2}
>>> update_sales(d, "infile1.txt")
>>> d
{'wallet': 6, 'phone': 8, 'gum': 20, 'keychain': 3, 'helmet': 4}
>>> update_sales(d, "infile2.txt")
>>> d
{'wallet': 6, 'phone': 8, 'gum': 33, 'keychain': 3, 'helmet': 6, 'paper': 1}
>>> update_sales({}, "infile3.txt")
Traceback (most recent call last):
...
ValueError: Item sales must be nonnegative.
>>> update_sales({}, "infile4.txt")
Traceback (most recent call last):
...
ValueError: Too much information on given line.
```

5.4 Part #4

On Canvas, you will find a file called `person.py` that contains the definition of `class Person`. You will *not* submit this file `person.py`. In `prog1.py`, you will need to `import` the class `Person` from `person.py`. *You will be penalized if you paste the definition of `class Person` into `prog1.py`.* For reference, below is that definition.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.best_friend = None
```

Implement a function called `read_people` that takes one argument, the name of a file, and returns a dictionary that maps strings to instances of the class `Person`. Below is a text file similar in format to the kind of files that `read_people()` will deal with.

infile1.txt:

```
Jacob 22
Stephen 24
Rachel 27
Mitchell 22
Angie 24
===
Angie: Rachel
Jacob: Stephen
```

More generally, the text file whose name is passed to `read_people()` will always have the following format:

1. List of people's information where each line should be used to create an instance of `class Person` that is put into the dictionary (with the person's name as the key) that is eventually returned by `read_people()`. For example, the above file would lead `read_people()` to generate a dictionary containing five key-value pairs. One of those key-value pairs would consist of the string `"Jacob"` mapped to an instance of `Person` with the `name` member set to `"Jacob"` and the `age` member set to `22`.
2. Three equal signs.
3. List of pairs where each line has a pair of two names (always separated by a colon and one whitespace). These pairs specify the best friendships among the people named, and you should set the `best_friend` fields of the corresponding `Person` instances appropriately. For example, upon processing the line `Angie: Rachel` from the above text file, `read_people()` should ensure that:
    - The `best_friend` member of the instance of `Person` corresponding to Angie references the instance corresponding to Rachel.
    - The `best_friend` member of the instance corresponding to Rachel references the instance corresponding to Angie.

You may assume that the input file is properly formatted. However, properly formatted information can still be invalid information; `read_people()` must `raise` a `ValueError` in any of the following cases. (The message associated with your `ValueError` will not be checked.)

- ***No imaginary friends***: A nonexistent person (i.e. one whose information was not given before the three equal signs) is assigned a best friend.
- ***No double-crossing***: A person's best friend is stolen from them, e.g. this would occur if one added the line `Mitchell : Stephen` to the end of `infile1.txt` above, because Jacob's (so-called) best friend Stephen would be stolen from him. This would also occur if the line `Rachel: Mitchell` were added, because Angie's best friend would be stolen from her.
- ***No "reaffirming"***: If, for example, Jacob and Stephen are considered best friends after the line `Jacob: Stephen` is encountered, then encountering another line reaffirming this best friendship (i.e. `Stephen: Jacob` or `Jacob: Stephen`) should also mean the input is invalid.

Your function must close any file that it opens. You may assume that each line of the text file ends in a newline character.

Below are interactions with `read_people()` on the Python interpreter, using the `infile1.txt` file shown above. Note that when an instance of `Person` is output, its address in memory will be printed (e.g. as in `<person.Person object at 0x7f9f8b125518>`); that part (i.e. the `0x7f9f8b125518`) need not be the same on your end.

```
>>> p = read_people("read_people_test_files/infile1.txt")
>>> p
{'Jacob': <person.Person object at 0x7f9f8b125518>, 'Stephen': <person.Person object at 0x7f9f8b125278>, 'Rachel': <person.Person object at 0x7f9f8b111278>, 'Mitchell': <person.Person object at 0x7f9f8b111ef0>, 'Angie': <person.Person object at 0x7f9f8b111f60>}
>>> len(p)
5
>>> p['Stephen']
<person.Person object at 0x7f9f8b125278>
>>> type(p['Stephen'])
<class 'person.Person'>
>>> p['Stephen'].name
'Stephen'
>>> p['Stephen'].age
24
>>> p['Stephen'].best_friend
<person.Person object at 0x7f9f8b125518>
>>> p['Stephen'].best_friend.name
'Jacob'
>>> p['Stephen'].best_friend.age
22
>>> p['Stephen'].best_friend.best_friend.name
'Stephen'
>>> p['Mitchell'].age
22
>>> p['Mitchell'].best_friend  # the output is not cut off here; the value is None, so the interpreter doesn't show anything
>>>
```