TU856/858-2 Object Oriented Programming with Python
Semester 1, Lab 3, 09.10.2020
Dr. Bianca Schoen-Phelan

Please join the virtual classroom in brightspace first, before going to individual lab links. I will provide explanations and hints and tricks regarding the lab at the beginning of the lab sessions. These are recorded.

Please sign your group's lab sign in sheet!
Lab solutions will be discussed in the Monday lecture. Labs are not marked.
Link to the lab groups (click).
Link to python documentation (click).

Objectives:

- New git/gitHub scenario
- Practice string operations
- Practice control flow statements

ONLINE guideline:

- We meet in the virtual classroom in brightspace first. Just like the last lab.
- You will be able to stay within your own lab group throughout this online lab. Each tutor offers a teams link. You will be required to sign a sign-in sheet for each lab group.

Task Background:

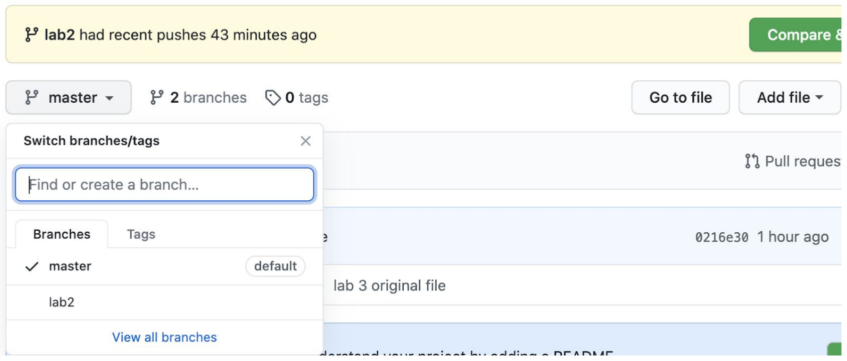
Read the following paragraph as quickly as you can, and see if you encounter any difficulties.

> **Aoccdrnig to rscheearch at an Elingsh uinervtisy, it deosn't mttaer in waht order the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer is at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae we do not raed ervey lteter by itslef but the wrod as a wlohe.**

This has been presented as an example of a principle of human reading comprehension. If you keep the first letter and the last letter of a word in their correct positions, then scramble the letters in between, the word is still quite readable in the context of an accompanying paragraph. However, it seems that this is a bit of a myth and not truly based on solid research. In short, for longer words the task is much more difficult. Nonetheless, we are going to imitate the process on some English text.

1. **Fetch and merge the lab3.py file from the BiancaSP/OOP2020-21 repository:**
   a. **Before** getting the new lab file, **decide** if you want to keep *your edited* version of the `lab2.py` from last week's lab or not. The course module github repo contains a new file `lab2solution.py` and a new file `lab3.py` (as well as the old files, of course). You cannot merge your local repo with the class's online repo containing these new files (after `git fetch upstream`) if your local version of an existing file (in this case `lab2.py`) is different to the version on the course repo; you would receive a conflict error message. If you are happy to overwrite your own `lab2.py` file and lose all your work from last week, you can follow the steps as outlined here, **otherwise go to (b)**:
   ```
   $ git fetch upstream
   $ git reset --hard upstream/master
   $ git merge upstream/master
   ```
   This will overwrite anything that happened with `lab2.py` (or the first lab) in the last lab.
   If you have pushed your updates from last week to *your own* GitHub repo last week, then you need to force *your own* GitHub repo to use the version from the course gitHub repo that you have just integrated locally before you can push the updates to `origin` that you have just downloaded:
   ```
   $ git push origin master --force
   ```
   You have now lost all traces of the work you did last week ;) and synched all repositories.

   b. If you want to keep your version of `lab2.py`, and you haven't created a new branch last week for your new work, **now** is when you need to create a new branch for your lab 2 work:
   ```
   $ git checkout -b my_lab2
   ```
   Check that you now have two branches:
   ```
   $ git branch
   ```
   This should list **master** and **my_lab2** with an * before my_lab2 to indicate that this is your current branch. You should reflect this change in ***your own*** online repo on GitHub:
   ```
   $ git push origin my_lab2
   ```
   This will create the new branch for lab2 in ***your own*** online GitHub repository. Check in your browser that you can see the new branch appear (there is a downward arrow icon next to the word **master** or **main**, click on this, and you should see the my_lab2 branch as an option, see image below)



In the terminal/gitbash/console return to the master branch:

```
$ git checkout master
```

Check that you are on the master branch:

```
$ git branch
```

You should see the same two branches, but the asterisk should now be in front of **master**. You may be told that there is a difference between your origin/master and your local master branch.

At this stage the `lab2.py` in your **local master** branch is still your edited version from the last lab. If you were to do a fetch and merge you'd get a message of a conflict between the course repo and your own repo and that you cannot merge the master branch (to receive the updates). Because we have just saved a copy of your own work in the branch my_lab2, it's ok to overwrite this version in the master branch with the original from the course repository:

```
$ git fetch upstream
$ git reset --hard upstream/master
$ git merge upstream/master
```

Check that you have the two new files: `lab2solution.py` and `lab3.py`

All your work from Lab2 is now saved in the branch my_lab2, in both your **local** repository and your own **online** GitHub repository. We can therefore overwrite your **online** master branch with the "original" version of lab2.py (that you just downloaded in the previous step) to synchronize your local master branch with *your own* GitHub repository by simply overwriting it:

```
$ git push origin master --force
```

This means that the version in the master branch of the files and the file versions on your own gitHub should be the same. You can check it with a git diff:

```
$ git diff HEAD:Labs/lab2.py origin/master:Labs/lab2.py
```

You shouldn't get an output here, which means there isn't any difference between the files.

If you want to continue working on your lab2.py file, do this locally on the respective branch. Then, when you have made a change, add and commit the change locally and to synch with your own gitHub repo push it to origin.

   c. Some git tips/explanation:
      i. `git reset --hard` will bring your local repository back to a state before the last git merge, to the most recent commit before the merge. This is helpful if you didn't really want to do a merge or are unhappy with your strategy of resolving a conflict (but you'll lose your work to the previous commit).
      ii. Going forward with the labs, you can choose if you see value in maintaining *your own* GitHub repository as a copy of the work that you are doing locally on your computer. For this course new files will be provided on the course GitHub repository `BiancaSP` every week, including a solution of the previous week's lab.
      iii. Please create a new branch for every new lab before editing the new lab's file using:
      `$ git checkout -b name_of_new_branch`
      It might be helpful to choose a branch name that reflects the lab week you are in, for example `lab3` for the name of the branch this week. Do all edits on this new branch. If you want to synchronize your local work with your own online GitHub repository, switch back to the master branch, and push the new branch to your origin as done above. Do not merge these branches with the master/main branch.

2. **Use the structure in provided lab file and write a Python program that scrambles text as above:**
   a. Make sure you started at the last point in (1) and have created a new branch for this lab before editing the `lab3.py` file.
   b. The text that is to be scrambled is provided via the `input()` function.
   c. **Do not** use any in-built functions like `random()` etc. You are supposed to solve the exercise using control flow structures and the data types we have discussed, mainly strings and lists, if statements and loops and casting of data types.
   d. Extra points for controlling input, i.e. checking if the right type of input has been provided.
3. **Hints:**
   a. Start with scrambling only one word. For example, 'Hello', scrambled at index 1 and 2 results in 'Hlelo', this preserves the start and end, as in the example above.
   b. Check that the input is of the correct data type, or cast/UPPER/LOWER/strip() the string already when it is inputted
   c. Scrambling only makes sense if you have a certain number of characters in a word, i.e. anything below three characters should not be scrambled

d. Then move on to scramble one full sentence, by taking the input and then going through it word by word. You might want to cast your string as a list for this, and then scramble each individual item by looping through your list. You can join the characters inside a word back together using join().
e. Then scramble one sentence including punctuation. For punctuation you can either use the provided `string.punctuation`, or manually look for “,”, “.” etc. If you try `print(string.punctuation)` you will get a list of what it contains: `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`
When you have achieved this, and would like to try a harder version, then consider mid-word punctuation, such as hyphens.
It is up to you how hard you want to make the programme. For example, the easy version is to just check for punctuation at the end of a word. The harder is to check for punctuation, such as hyphenation of words, etc.
For an overview of string functions see https://docs.python.org/3.8/library/string.html

Well done!