F. Wermelinger
Office: Pierce 211

Homework 2

Git remotes, Basic Python, Closures, Decorators, Matplotlib

Issued:	September 13th, 2022
Due:	September 27th, 2022

Note this is a 2 week exercise. *Do not procrastinate the work.*

Problem 1: Homework Submission Requirements (*10 points*)

Requirements:

1. Complete the homework on the designated branch (***5 points***)
2. Create a pull request to merge the designated homework branch into your default branch, e. g., `main` or `master` (***2 points***)
3. Merge the open pull request of the *previous homework* into your default branch **after** you have received feedback from the teaching staff and all issues are resolved (***3 points***).

Deliverables: your solutions to the problems below must be submitted in a directory called `submission`.

See `https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-hw`.

Problem 2: Git Remotes (*15 points*)

Deliverables:

1. P2_config.png
2. P2_log.png
3. P2_mergelog.png

In the previous homework you already worked with remotes. In this problem we will study remote repositories a bit more. This problem will follow the same directory structure suggested in the previous homework. You are free to implement your own choice. Be sure that the directories `local` and `bare` are not inside any Git repository. This problem will follow this structure:

```
classes/
|-- CS107
|   |-- git
|   |   |-- myrepo <- your private Git repository
|   |   |-- sandbox <- forked sandbox from previous homework
|   |   |-- hw2
|   |   |   |-- bare  <- bare repository for this problem
|   |   |   \-- local <- local repository for this problem
|   |   \-- main
|   \-- CS107_other_data
\-- CS205 (some other class in this directory)
```

a) ***5 points***

Following the introduction above, create two directories `bare` and `local` which are not inside a Git repository. These directories will become two Git repositories themselves where the former will be a *bare* repository and the latter a regular Git repository as you are familiar with up to now.

Change into the `bare` directory and initialize a new Git repository with the familiar command

```
$ git init --bare
```

The additional option is new and will create a bare repository. Type `ls` to inspect the directory contents.

Change into the `local` directory and initialize a new Git repository with

```
$ git init
```

Type `ls` again to inspect the contents. The directory *is not* empty, instead type `ls -a` and you will find that Git has created the `.git` hidden directory. Type `ls .git` and compare with what you observed in the bare repository.

A bare Git repository only contains the contents of `.git` in a regular Git repository. That is, all VCS information is stored in `.git` in Git. If you delete this directory in your repository, you will *lose* the version control and your repository becomes a normal directory with files in it.

Finally, compare the different repository configurations using

```
$ diff bare/config local/.git/config
```

which assumes that it is executed from the parent directory. Study the two different configurations. Take a screenshot of your terminal and save the file as `P2_config.png`

b) ***5 points***

Navigate inside your `local` Git repository. Copy the license text at `https://mit-license.org/` and paste it in a file called `LICENSE` in the root of your `local` repository. Replace the `<copyright holders>` place holder with your name. Create another file in the same location called `README.md`. Write a brief description what this repository is about and save the file.

Use `git status` to check your repository status. Add both files to the index and create a commit with a descriptive commit message. Check your history with `git log` and create a screenshot of the output. Save it as `P2_log.png`.

You have not added any remote repository at this point so you can not push your changes. You now want to use your bare repository as a *remote* since you only intend to use it as a *hub* for pushing and pulling commits. You can do this with the familiar command

```
$ git remote add origin ../bare
```

Note that this assumes that the command is executed in the `local` repository and the directory structure is as described in the problem introduction. You can use any other path for the last argument otherwise. Note that we have not used an URL this time, Git can also work with local paths. This is how remote repositories are stored on sites like GitHub for example. Since these remotes do not need to checkout a working tree, bare repositories are used instead.

Finally, push your commit to the newly added remote.

c) ***5 points***

Navigate into your *forked* sandbox repository from last homework. You should still be on the `hw1p5` branch, if not checkout this branch. Add the local `bare` repository as another remote and name it `bare`. You should now have three remotes defined for that repository. Test it with `git remote -v`.

Finally, you want to *merge* the default branch in the `bare` remote you have just added into your `hw1p5` branch of your forked sandbox. First you need to `fetch` the changes in the `bare` remote. You can do this with

```
$ git fetch bare
```

Alternatively, you can use `git fetch --all` to fetch all defined remotes at once. Now use the `git merge` command to merge the default branch in the `bare` remote into your sandbox fork. Before you merge, think about the possibility of a merge conflict and where it could happen. After you have concluded the merge, take a screenshot of `git log` and save the file as `P2_mergelog.png`.

Ensure that you have copied the deliverables into your `submission` directory in your private Git repository.

Hint: *Because your sandbox fork and the bare remote are unrelated repositories, Git may refuse this merge. You can use the* `--allow-unrelated-histories` *option to enforce it.*

Problem 3: Python Basics — DNA Complement (*20 points*)

Deliverables:

1. `P3.py`

Important: this problem will be partially *auto-graded*. You are expected to work with the skeleton code provided in `code/p3/P3.py`. Do not change existing code in that skeleton code. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.

Much of biology and bioinformatics depends on understanding and manipulating long sequences of DNA. A common thing to look at would be the complement of a given sequence. Recall that DNA has 4 bases: A, T, G, C. DNA sequences are described by a concatenation of these letters. For example, a DNA sequence might have the form AATGGC. The complement of the base A is the base T and the complement of the base G is the base C. The DNA complement of AATGGC is therefore TTACCG.

Write a Python function that accepts a string of arbitrary length representing a DNA sequence and returns the corresponding DNA complement also as a string. Write this function in a file called `P3.py`.

Your implementation must have the following requirements:

- Your function must be named `dna_complement` and have it take in a DNA sequence as input string.
- You must handle the situation in which the input string is either empty or does not contain characters corresponding to the real DNA bases. If either of these situations arise, your function should return the type `None`.
- Your function should be case-insensitive. This means that the input string to the function can be lower, upper or mixed case. However, your function must return the DNA complement in upper case.
- Provide a demo of your implementation. You should do this in the same file in which the function is defined. The demo should include the following:
  - Print out an example input string with *mixed* case letters. You can choose any combination you want.
  - Call your implemented function with the above string as input.
  - Print the output string that your function returns.
- Repeat the demo with an invalid DNA string (containing characters other than A, T, G, C).

Hint: *Assume that the DNA sequence is provided in the code. There is no need to have the user input the string on the command line.*

Problem 4: Closures and Decorators in Python (25 *points*)

Deliverables:

1. P4a.py
2. P4b.py
3. P4c.py
4. P4d.png

The goal of this problem is to write a simple bank account withdraw system. The problem is based off of a problem in *Structure and Interpretation of Computer Programs* by H. Abelson and G. Sussman. Solve each part in a different file as indicated by the deliverables above. Please use comments and meaningful naming in your code.

a) ***10 points***

Write a closure to make withdrawals from a bank account. You *must* use a nested function for this problem.

The outer function should accept the initial balance as an argument (we will refer to this argument as `balance` in this problem statement, but you can name it whatever you think is more meaningful. The inner function should accept the withdrawal amount as an argument and return the new balance as a numeric type.

Your implementation must check for the case where the user tries to withdraw more than the current balance holds. If you detect such a case, use the `raise` statement[1] to raise an exception. For example

```
raise ValueError(f"Amount of {amount} exceeds balance {balance}.")
```

After you have implemented your code, write a small test *in the same file* with two consecutive withdrawals, for example

```
wd = make_withdraw(initial_balance)
print(wd(withdraw1))
print(wd(withdraw2))
```

You should observe that the behavior is not correct. *Add an explanation* to the file P4a.py why this is the case. Use the `print` function for your explanation at the end of your test code. For example:

```
print("My explanation")
```

Hint: *Do not try to assign* `balance` *a new value in the inner function for this problem (you will try this in the next task). Focus on the base implementation and test code.*

[1] https://docs.python.org/3/reference/simple_stmts.html#the-raise-statement

b) ***7 points***

Implement a fix for your previous code such that consecutive withdrawals produce a correct balance. Explain why this fix does not work. See `https://docs.python.org/3/reference/executionmodel.html` for additional helpful information when you try to formulate your explanation.

Write your code in `P4b.py` and use the `print` function for your explanation, similar as in task 4a.

c) ***5 points***

Provide a fix for the name resolution issue encountered in task 4b and test your code again similar to task 4a.

Hint: *Have a look at the* `nonlocal` *keyword. See* `https://docs.python.org/3/reference/simple_stmts.html#nonlocal`

d) ***3 points***

Finally, visualize your code from task 4c using the Python Tutor at `https://pythontutor.com`. Take a screenshot *at the final step* and name it `P4d.png`.

Problem 5: Analogue Clock in Python (*30 points*)

Deliverables:

1. `P5a.py`
2. `P5a.png`
3. `P5b.py`

Important: task 5a will be partially *auto-graded*. You are expected to work with the skeleton code provided in `code/p5/P5a.py`. Do not change existing code in that skeleton code unless noted in local comments. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.

The hands of a clock can be defined by two points. The first point is given by the origin $(0, 0)$. The second point will be somewhere on a circle. Precisely where on the circle depends on the time of day and the clock hand that we are thinking about (i. e., hour, minute or second). The time of day can be converted to the degrees of a circle (or an angle). For example, we are used to looking at a clock with 12 at the very top of the circle. In our minds, this corresponds to 0°. From this perspective, 3:00 in the afternoon corresponds to 90°. A few hours on the clock correspond to the following angles (in degrees):

Time	12:00	3:00	6:00	9:00
θ (degree)	0	90	180	270

From here, you should be able to see that each hour represents 30°. And so, in degrees on a clock, we have, $\theta = 30t_h$, where t_h is the hour. Of course, this is at odds with the usual mathematical definition. In mathematics, we would expect 3:00 to be at 0° and 12:00 to be at 90°. We can introduce a shift to fix this $\theta = 90 - 30t_h$. Think about this transformation for a second. There is one more little modification that can be done, which will help the readability of the hour hand. Since each hour consists of thirty degrees, we can have the hour hand move a little bit each minute to help it slide along between two hours. This can be accomplished by recognizing that there are 60 minutes in an hour. Hence, at the first minute of an hour, the hour hand should be pointing directly to the hour that just started. At minute 60, the hour hand should be pointing to the next hour, which means that it will have rotated by 30°. Hence, the angle of the hour hand is given by,

$$\theta = 90 - 30t_h - \frac{t_m}{2}.$$

where t_m represents the minutes.

Now that you know the angle that the hour hand has moved through, you are just about ready to calculate its (x, y) position. Remember, this is the second point on the line. You just need to do two things:

1. Define the length of the line by some parameter r. You choose this parameter. It just tells you how big the hour hand is.
2. Convert θ to radians: $\theta_h = \frac{\pi}{180}\theta$

Then the (x, y) coordinates are given by,

$$x = r\cos(\theta_h),$$
$$y = r\sin(\theta_h).$$

Now you are ready to plot a line representing the hour hand.

The minute and second hands are much easier. The angles that the minute hand and second hand pass through are

$$\theta = 90 - 6t_m$$

and

$$\theta = 90 - 6t_s,$$

respectively, where t_s are the seconds. You can compute the (x, y) coordinates of each hand the same way as you did with the hour hand. Just choose a length r and convert degrees to radians.

a) ***20 points***

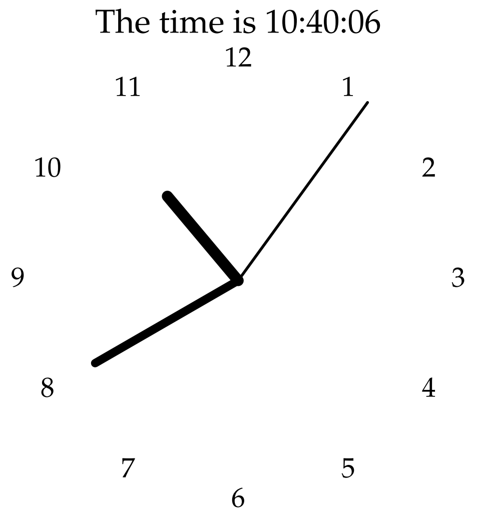

FIGURE 1: It is time to learn Python.

Write a closure with the following API:

- The outer function should take in a single floating point number representing the length of the clock hand. This is parameter r in the description above.
- The outer function should return an inner function.
- The inner function should take in a single floating point number representing the angle of the clock hand. This is θ in degrees in the math above.

- The inner function should return the (x, y) coordinates of the clock hand on the circle.

Start your implementation of the closure using the file provided in `code/p5/P5a.py`. See the comments for further hints. A possible plot could look like the one shown in figure 1. Save your plot in the file `P5a.png`.

Hint: *Do not forget to convert degrees to radians.*

b) ***10 points***

You created an analog clock. Sadly, it only displays the current time. It would be cool if you could make the hands move. In this task of the assignment, you get to animate the clock. Follow these steps:

- Write a loop of your choice and plot a new clock at each iteration. This means you will need to get the current time at each iteration of the loop. It is up to you how to terminate the loop. *The loop must not be infinite.*
- You can use `plt.cla()` or the `clear()` method of an axes object to clear the axes.
- Use `plt.pause()` to add a small pause and give the CPU some rest.
- You can use `fig.canvas.draw()` to interactively update the plot. Here `fig` is a `plt.figure()` object.

Write your code in `P5b.py`. You may think about code reuse from task 5a.