T04: Adventures in Gitland

- Recall our discussion on Day 1 about how to work well with your partner.
- Pair programming involves two roles: the driver (who types the code) and the navigator (who reads the instructions).
- If you run into an issue, then work with your partner to solve it.
- If both you and your partner have the same issue, then summon a teaching assistant or instructor.

Learning Objectives

- Refactor existing code into functions.
- Get additional practice with nesting conditionals.
- Learn to use git to collaborate.

How to Start

- To begin, make a copy of this document by going to File >> Make a Copy...
- Share the copied document with all members of your team. You can share this document by hitting the blue button in the top right of the document, then entering the email addresses of all members in the bottom input field.
- Change the file name of this document to **username1, username2 - T04: Adventures in Gitland** (for example, **pearcej, heggens - T04: Adventures in Gitland**). To do this, click the label in the top left corner of your browser.
- We will **not** be using GitHub Classroom for this assignment. Instead, use PyCharm to clone the code from the T04 master repository. ***Don't start editing code until we instruct you to do so!***
- First, discuss with your team and assign yourselves roles.

Github Repo Link:	https://github.com/Berea-College-CSC-226/t04-main

First, discuss and assign roles. *Try to pick the role you've had the least experience in.*

Driver[1]:	
Navigator[2]:	
Quality Control[3] (if the group has three members):	

---

[1] The driver will be doing the majority of the typing in PyCharm. Your job is to solve the problem given to you by the navigator.

[2] The navigator will be giving directions to the driver, and helping the driver catch syntax and logic errors as he or she creates the code. The navigator should keep track of time and make sure progress is being made.

[3] The quality control specialist will ensure rules are being followed, both in the code (suggesting places to add comments, watching for misspellings, etc.) and in this document (making sure the questions are being answered at the right times, checking for typos, etc.) In a group of two, everyone is responsible for quality control.

A long time ago…

...in Teamwork T01, remember when we created a text-based adventure game where the user decided what happens in the story?

In the T04 repository that you cloned, you will find **t01_final_story.py**, which is the product of your T01 creation. As much as we'd like to enjoy the story now that it is all compiled... it has errors!

One of the many useful skills you'll need to be a programmer is the ability to **refactor code**. Refactoring code is a process where we take code that is already written, and make it better.

So why do we need to refactor T01? Well, there are a number of reasons:

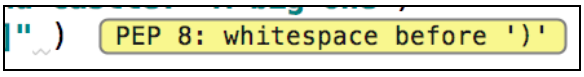
1. Since T01, you've all become much more knowledgeable coders, and you may notice that your code has some design flaws that you now know how to fix. And bugs!
2. Now that you know about functions, you can see the structure of T01 is all wrong per our new rules (i.e., having a `def main()`; no code at the top level (no indenting); no mental blocks encapsulated into functions, etc.).
3. This code is broken!
4. This code is kind of a mess!

Discuss with your partner two or more ways in which you've improved as a coder since T01. You can refer back to **t01_final_story.py** for "inspiration".	2. We feel more technical with our code, we know more about what we are doing, we can detect errors quicker and even evade when coding, and we are more organized with all of it.

"Better code" can mean a lot of things: easier for the programmer to read; more efficient in terms of lines of code; more efficient in how well it runs; more abstracted, so the code can be reused elsewhere.

You might still be asking, but WHY do we need to refactor T01? Here are two even more compelling reasons:

1. When you leave Berea College and get your fancy job as a programmer, you'll be expected to conform to your company's coding standards (i.e., how your code looks so everyone else in the company can easily read your code). For Python, many companies follow the de facto PEP 8 Standards, and customize them to fit their exact needs. You've likely noticed many gray squiggly lines and warning messages about PEP8 in PyCharm:

   Refactoring code is one method you'll need to be familiar with to be an effective programmer, no matter who you work for in the future. ***Every programmer has to refactor code!***
2. By refactoring the T01 code, you're going to improve your skills with writing functions, writing useful documentation, and also learn how to use git to collaborate with the rest of the class. So, making T01 pretty isn't really the goal of this assignment; learning these other **essential** skills is the ultimate goal.

Your Tasks

For this teamwork, we'll be using git to manage our code. A large part of this teamwork is understanding how git works, and how git facilitates multiple teams of programmers working on one set of code, without clobbering

each other's work. At a high level, we will clone the code from a repository on Github, make a new branch, make edits to the code, and push those changes back up to Github. Things should get interesting as multiple teams finish their changes and try to push them to the repository. We'll see what happens…

Branching

So far, we've not used git for collaboration very heavily. In this teamwork, however, everyone in the class will be modifying the same starter code…

To start, create and checkout a new branch. To do this in PyCharm:

1. Right click on the **t04-master** repository folder, then do **Git >> New Branch.** Name the branch you're about to create your username(s) (e.g., **pearcej_heggens**). Make sure that you are working *on your team's branch* before proceeding.

Inside the **t04_refactored.py** code, you'll notice a function definition for every team:

```
def team_1_adv():
    pass
    # TODO Add your code here
```

2. Copy your team's code from **t01_final_story.py** and paste it into your function. Watch the indentation!
3. Read through your code, and make changes to it when you notice places that could use improvement. Remember, your classmates wrote that code several assignments ago… see how much we've grown as programmers already!
4. Look for gray squiggly lines, indicating places where PEP8 standards aren't being followed. Make the suggested change by PyCharm to remove the squigglies.
5. In your function's docstring, add a link to your team's Google Doc (this document) as well as the names of all partners who worked on that function. For example:

```
def team_1_adv():
    """
    https://docs.google.com/document/...
    Scott Heggen
    Brian Schack
    :return: none
    """
```

Briefly describe any logic changes you made in your code, and why you made them.	3. We made logic changes in HasSword and goblinhead. Basically we changed the "== true" after HasSword and golblinhead to only HasSword and "if not HasSword:". Same thing with goblinhead
Did you find it challenging following the logic of	4. In some way. Everyone thinks and code differently

another group's code? Why or why not?	but after reading code day by day, it is getting better following the logic.
Briefly describe two or more of the PEP8 warnings you fixed. Does the resulting code look better or worse? Why?	5. PEP 8: E265 block comment should start with '# ' PEP 8: E303 too many blank lines (2) These were two warnings we fixed. It does look better and more readable because there is some space between line of codes.

Commit and Push your Changes

Test your code and make sure nothing is broken. Be sure you check ALL combinations of inputs and paths through your logic. When you're confident it's ready to go to the repository, **commit the file** (don't forget to write a meaningful commit message!), and **push** the changes to the repository.

Go to GitHub in your browser, and take a look at the repo.

You should notice that there are multiple branches in the Github repository. Find yours and check that your code is there. Compare your branch to master. What's different?	4. ?
Do you see other groups' branches in the repo as well? Is your code in their branch? Why or why not?	5. There were the branches that were made from other team and we were able to see the code of the other teams and others can view our branch on Github.
Next, issue a **pull request** (this is done in the browser on Github). New pull request A pull request is a formal request to add your code into the master branch, for all to see and share. Once you've issued the pull request, communicate to the instructor that you are "ready for a PR review." After the instructor or TA approves your request, refresh Github. Is your code in the master branch now?	6. INO. But the we have been approved for the PR.
What about other groups' code? Is it also in master branch, compared to the last time you looked at it in Question 5?	7. NO.
Once your code has been pulled into the master branch, go back to PyCharm. Right click on the directory, and switch back to the master branch by	8.NO.

clicking Git >> Branches and selecting “Local Branches” >> Master”. Is your code in your **local** master branch? Why not?	

When you issued the pull request in Github, you (in PyCharm) became out of sync with the repository in Github (i.e., your local is behind the remote). You need to synchronize. To do this, you need to **fetch** changes from the remote. Go to **Git** >> **Fetch** to get all of the changes occuring in the remote repository (i.e., Github). You may see lots of other groups’ branches now, depending on how quickly you’re working through this assignment. You can fetch multiple times to see updates as master changes.

However, fetch only grabs *references* to each branch; the code hasn't been pulled into your local computer yet. **Git** >> **Pull**... allows you to actually bring the code from Github to your machine.

When you **pull**, Git does its best to **merge** the remote changes into your local changes. If there are conflicts, PyCharm will warn you. More on that later... if you get into this situation, ask for help.

Finally, PyCharm did some work to help simplify this process by combining these three steps into a single button: Git >> Update Project. I don't mind if you use this button, but you do need to understand the three operations described above: **Fetch, Pull, Merge**!

Git Terminology

Git can be challenging to wrap your head around when you’re first learning it, especially when collaboration becomes a part of the process. So, let’s make sure we understand the basic terms; the details of using git will be touched upon again in your homework assignment, in more detail.

Define each term:

Clone	9.a. A clone is a copy of a repository that lives on your computer instead of on a websites server somewhere or the art of making a copy.
Commit	9.b. Or a “revision” is a individual changeto a file. When making a commit Git creates a unique id and that ex. Hash and that allows you to keep records of the specific changes along with who made them and when.
Push	9.c. A git pushes that overwrites the remote repository with local changes without regard or conflicts.
Branch	9.d.A branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or main branch allowing you to work freely without disrupting the "live" version.
Pull	9.e.Pull refers to when you are fetching in changes and merging them.
Pull request	9.f. Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators.
Merge	9.g. Merging takes the changes from one branch (in the same repository or from a fork), and applies them into another.

Remote	9.h. This is the version of a repository or branch that is hosted on a server, most likely GitHub.com.
Local	9.i. a copy of the entire project's history and codebase that resides on a developer's machine.
Fetch	9.j. When you use git fetch, you're adding changes from the remote repository to your local working branch without committing them.

Check your definitions against the git glossary to ensure they are correct.

In Github, go to the Network Graph (i.e., a history of branching and committing history). Discuss with your partner what confusions you still have about the git workflow you experienced today, using the network graph to explore what has happened. Write your unanswered questions in the space to the right:	10. With gitworkflow there was not anything that we were finding troublesome. But with navigation we are getting more better at findin the pull request link and seeing the changes that we need to make for the code to run properly. And most difficult part we had during this whole teamwork is the finding of the mistakes in the code and making it work properly.

Submission Instructions

At the end of every assignment, I will include these instructions. They do change on occasion, so be sure you check them each assignment to ensure no special instructions were added.

Follow the submission instructions by Friday at 11:55PM.