⚔ Adventure game

A. Project Overview

In this project, you'll make a simpler version of an old-fashioned text-based adventure game. You can find an example in the workspace below. Try it out to get a feeling for how the game works!

In order to build a program like this, we should first completely understand what it does. Take a notepad out, and play the game multiple times. The game will present you some scenarios and ask you to make one of 2 choices, by entering `1` or `2`. In your notepad, record what happens each time you make a certain choice.
Note: Before doing this project, be sure to have completed the *Style and Structure* lesson in *Intro to Python, Part 2.*

Task List

- ☑ To try the example, run `python3 adventure_game.py` in the below workspace.
- ☑ Play the game at least a couple of times; you'll see that some aspects of the game are different each time.

REPL

No Open Terminals

NEW TERMINAL

Menu

Expand

As you can see, this is only a very short game with a couple of choices available to the player. We did that on purpose, to keep it simple. If we were making a complete game, we would expand it a great deal, and probably give the player a whole world to explore. But for this project, the idea is to focus on some key things that we need if want to make a working game:

- The game gives players a description of what's happening, and then asks them to make a choice.
- Something different happens depending on the choice the player made.
- The game also includes some random factors, so that it's a little different each time.
- The game has conditions for winning and losing.
- When the game is over, it asks if the player wants to play again.

These are the key features that your project will need to have in order to make it into a playable game. We'll go over each of them in more detail below.

As long as your program does all of the things listed in the instructions below, you are free to be as creative as you like!

Workspace or Local Environment

We've provided a workspace that you can use to complete this project (see the page titled **Project Workspace**. We suggest that you open these instructions in a second tab or window so you can have them alongside the workspace as you write your code.
If you prefer, you can also write your code on your own machine and copy it into the workspace when you're ready to submit.

B. Project Rubric

When doing a Udacity Project, it's an excellent habit to view the rubric once before doing the project *(to understand the requirements)* and once after doing the project *(to check your work)*. You can go to the **Project Rubric** page to read the details. For your convenience, the next page also describes the rubric.

C. Project Instructions

1. Print descriptions of what's happening for the player

One thing the game will need to do is to print messages for the player to describe what is happening. Like at the start of the example game, these lines are printed:

```
You find yourself standing in an open field, filled with grass and yellow wildflowers
.
Rumor has it that a wicked fairie is somewhere around here, and has been terrifying the
nearby village.
...
```

This is a very small step, but a good way to get started!

Task List

- ☐

  In your code editor, make a new `.py` file.

- ☐

  Add code to the file so that it will print a few messages for the player, each on its own line. You can use the same lines the example game uses when it starts.

- ☐

  Test the code by running your program from the terminal.

2. Pausing between printing descriptions

Now, one thing you might have noticed is that when you have multiple `print` statements, they get displayed in the terminal so fast that it looks like they all just show up at the same time. That's no good.
If you look at the example game, there's a short delay after each line is printed.

You may remember that earlier in the course, we briefly introduced the `time` module and the `time.sleep()` function:

```
import time
print("Hello")
time.sleep(2)
print("world!")
```

This will cause a 2-second delay after the first `print()` statement. Give it a try with the messages in your code!

Task List

- ☐

  Use `time.sleep` to create a delay between messages. If you like, you can experiment with delays of different lengths for dramatic effect.

- ☐

  Run your code to check that it still works. (It's a good idea to do this frequently while you work!)

3. Give the player some choices

Up till now, our program prints the introduction of the game, with short pauses in between each sentence. Another important element of any good adventure game is *choice*. To do this, you'll need to get some input from the player, and then change what happens depending on what that input is. In the example game, that looked like:

```
Enter 1 to knock on the door of the house.
Enter 2 to peer into the cave.
What would you like to do?
(Please enter 1 or 2).
```

Task List

- ☐

  Present the player with a choice, and collect their response using the `input` function.

- ☐

  Have something different happen depending on the player's choice.

In the example game, each time we make a choice, something happens, and we are offered 2 choices again till we win or lose. For now, just focus on writing code for the 2 choices you have.

- If the player knocks on the door of the house, what happens?
- If the player enters the cave, what happens?

We will be dealing with subsequent choices after the first choice in the upcoming steps.

4. Make sure the player gives a valid input

Up till now, our program prints a description of the game-world to the player, gives them a choice, and prints what happens depending on their choice. An important thing to notice in the example game is that if the player enters something other than `1` or `2`, the game keeps asking them for a `1` or `2`. We don't want the game to accept invalid input, like `75` or `foo`!

```
(Please enter 1 or 2.)
75
(Please enter 1 or 2.)
foo
(Please enter 1 or 2.)
```

If the player tries to respond with something invalid, your program should ask them to try again—and it should keep asking them to try again until they give valid input.

Task List

- ☐

  Add code to check the input to see if it's valid. If they entered an invalid response, it should ask them to try again.

Recall the data type in which the `input()` functions stores the user input, and add code accordingly.

5. Add functions and refactor your code

By now, you may have noticed that some of your code is getting pretty messy, and some parts may be kind of repetitive. If you haven't already, this would be a good time to consider defining some functions, and moving some of your code into those functions.

For example, you probably have a lot of code that looks like this:

```
print("You find yourself standing in an open field, filled with grass and",
      "yellow wildflowers.")
time.sleep(2)
print("Rumor has it that a wicked fairie is somewhere around here, and has",
      "been terrifying the nearby village.")
time.sleep(2)
print("In front of you is a house.")
time.sleep(2)
print("To your right is a dark cave.")
time.sleep(2)
print("In your hand you hold your trusty (but not very effective) dagger.")
time.sleep(2)
....
```

This is quite repetitive—we are using `print`, `sleep`, `print`, `sleep`, `print`, `sleep`, over and over ... One improvement that you can make is to define your own function so that it will both print and sleep—you might call it the `print_pause` function.
With such a function, you could simply pass it a message to print, and it would take care of the pausing:

```
print_pause("You find yourself standing in an open field, filled with grass",
            "and yellow wildflowers.")
print_pause("Rumor has it that a wicked fairie is somewhere around here,",
            "and has been terrifying the nearby village.")
print_pause("In front of you is a house.")
print_pause("To your right is a dark cave.")
print_pause("In your hand you hold your trusty (but not very effective)",
            "dagger.")
....
```

Here's another way you can use functions in a game like this—you can define a function for each place the player can go. In the example game, the code looks something like this:

```
def fight():
    # Things that happen when the player fights

def cave():
    # Things that happen to the player goes in the cave

def field():
    # Things that happen when the player runs back to the field

def house():
    # Things that happen to the player in the house
```

That way, when the player chooses to go to one of these places, you can simply call the function that displays the description and choices for that place. This is especially good if you want the player to be able to go back to the same place repeatedly, from different locations in the code. In the example game, the player can get back to the field from both the house and the cave, and having the `field` function makes this much easier.
Usually, each function will print what happens after the player takes a certain choice, then offer another choice, and call the specific function depending on the choice the player makes. In the end, you would want to have a `play_game()` or `main()` function, which starts the game.

These are just a few examples of how you can use functions in your project. You'll probably find other ways you can clean up the code and reduce repetitiveness by defining (and calling) functions. (In fact, it's possible to do this project with almost every line of code being placed inside a function definition!)

Task List

- ☐

  Add some function definitions to your code to help organize it better.

6. Use randomness in your game

Another key feature of most games is randomness or chance. If everything always happens exactly the same way, it can become boring and predictable.

There are all sorts of ways you could use randomness in your game. Here are just a few possibilities:

- In the example game, the enemy creature is selected randomly each time they play. Sometimes it's a pirate, sometimes it's a troll, and so on.
- You could do something similar to randomize which weapons or magical items the player encounters.
- You could include a combat simulation in which the player and enemy deal random amounts of damage to one another (you may remember that we did something like this earlier in the course).

All of these can be done using Python's `random` module and the `random.choice` and `random.randint` functions that we learned about earlier. For example:
At the start of each game, you can set the random enemy creature.
There are countless other possible ways to use randomness in your game—feel free to be creative here.

Task List

- ☐

  Add some randomness to your game, so that the player sees something different each time they play.

- ☐

  Test your code with different scenarios. Check for any errors that happen.

7. Create *win* and *lose* conditions

Eventually, the game should come to an end—and tell the player that they won or lost.

The end result of the game should be influenced by the player's choices (and possibly some degree of randomness as well). Generally, it's a good idea to use randomness to

only *partially* influence the outcome. If what happens to the player is completely random, the player will feel out of control and probably won't enjoy it.
For example, in our example game, the player fights the enemy creature. If they win the fight, they win the game!

Task List

- ☐

  Create conditions to end the game, and tell the player that they won or lost.

8. Check if the player wants to play again

When Python gets to the end of your script, it will exit back to the terminal. But that's not a good player experience. Instead, it would be better if the game asked the player whether they want to play again:

```
GAME OVER

Would you like to play again? (y/n)
```

Just like with other choices you've asked the player to make, this will need to get the player's input and then check if the input is valid.

If the player indicates that yes, they want to play again, then the game should start over. Depending on what you added to your game, there may be some things that need to be reset. For example, if your player has a health score or they picked up magical items, these should go back to the way they were when the game starts over.

For example, in our example game, after the player fights the enemy, they are asked if they want to play the game again irrespective of whether or not they won.

Task List

- ☐

  Add code to your program so that it checks whether the player wants to play again.

- ☐

  If the player says they do want to play again, the game should start over from the beginning with the original setting.

9. Check your style with `pycodestyle`

When you're all done with your program, be sure to check it using the `pycodestyle` tool—and then fix any issues it raises.

```
pycodestyle adventure_game.py
```

When you're all done, running **`pycodestyle`** on your program should give no results (no news is good news!). **Here** is a nice website you can go through to understand the PEP8 style guide.

In any of the errors, it'll tell you the row, and column number of where the error occurs. you can just google on how to fix each, but here are some examples.

- `adventure_game.py:6:1: W191 indentation contains tabs`: It's saying that on line 6, column 1, the indentation contains tabs. To fix this, just replace all tabs with 4 spaces using the find and replace option in your code editor.
- `adventure_game.py:7:38: E225 missing whitespace around operator`: This error occurs when we don't leave a space between an operator and the numbers/strings on both sides. For example, instead of writing `print(2 + 2)`, we wrote `print(2+ 2)`. This can be fixed by having space on both sides of the operator.
- `adventure_game.py:12:47: W291 trailing whitespace`: This error occurs when we have some trailing whitespace at the end of the line of code. Just clear that extra whitespace at the end of each line of code.
- `adventure_game.py:21:80: E501 line too long (92 > 79 characters)`: Make sure each line of code is less than 79 characters. If you are printing a sentence longer than that, split the line into two using a comma, like this:

```
print_pause("You find yourself standing in an open field, filled with grass",
            "and yellow wildflowers.")
```

  Also, note how the lines separated by a comma are indented equally inside the `print_pause` function.
- `adventure_game.py:23:9: E101 indentation contains mixed spaces and tabs`: This means that on line 23, column 9, you are using a mix of spaces and tabs for indentation. Again, just replace all tabs with 4 spaces using the Find and Replace option in your code editor.
- `adventure_game.py:100:1: E305 expected 2 blank lines after class or function definition, found 1`: This is quite descriptive. Just leave 2 lines between function definitions.

As you can see, the errors are very descriptive so you won't need google all the time to figure out how to fix such errors.

Task List

- ☐

  Run `pycodestyle` on your game and fix all the issues it raises.

10. Test Your Code

Before submitting your project, be sure to test at it. Play through it a few times, checking to make those different choices all work, and that invalid input doesn't cause the program to crash. If you have built a game exactly like the example game, you can run the same scenarios you ran before starting to check if it works the same as in the example project.

Task List

- ☐

The game plays from start to finish without crashing or throwing errors.

If you've done everything on this page, your code should be in good shape for a review! But if you'd like to be extra thorough, you can check out the **rubric** your reviewer will be using to grade your submission.

Standout Suggestions

Tips to personalize your project submissions and make them stand out:

- Identify the repetitive parts and refactor them into functions.
- Reduce redundancy by applying the **DRY principle**.
- Use intuitive function names and avoid generic names to provide clear context.
- Avoid using global variables in the functions.

NEXT

;