MERCHANT

A Python Trading Game

Difficulty: **Intermediate**

Scratchpad Papanui
Oliver Coates
Version 1.0 - 28/10/2025

About this project

In MERCHANT, you will learn how to make a trading game about sailing between ports, buying and selling cargo, and upgrading your ship.

This project is intended for those who have learnt the basics of python and want to put their skills to the test creating an interesting game.

This project will have many **mandatory** tasks that you must complete, and additional **optional** tasks which you can choose to complete, depending on your skill level and creative passion.

Table of Contents

1. Getting Started

To begin our project, we will first create a folder called Merchant on our computer. This is where our game will be run by. This folder will be referred to in this tutorial as the **root folder**, or **root**.

Inside of the root, create another folder called code. This folder will hold all the python code for our game.

Then a file in the code folder called main.py, which holds the following code:

```
1
2   print ("Welcome to MERCHANT")
3
4   input("\nThe program has finished. Press any key to exit.");
```

This lets us know that our code is running.

By default, on windows, a python application will close the moment the code has finished running. This makes it difficult to develop as the program will close before we can read what has happened. The input() on line 4 exists to hold the application once it has finished. Make sure that this line is always the last line in the main.py - so all code you write should go in between lines 2 and 4.

Next up, create a file in the root and rename it to 'run.bat'. This will turn it into a batch script, which is a special file which runs a given command on windows. Right click on 'run.bat' and open it in your code editor and enter the following code:

```
1   py Code/main.py
```

This tells python to run our main.py python file which is saved under the Code folder. Depending on your python installation, you might need to change the **py** to be **python**, or **python3** if you are getting errors.

Try run your python script by double clicking run.bat and check that the above python code runs. Note that this will only work on windows devices, if you are using a mac, talk to a supervisor.

Now it is time to set up our IDE to work with our python project. For this document, we will be using Visual Studio Code (VSC). You are welcome to use IDE for this project, but the instructions won't line up.

First, open up VSC and go File > Open Folder, and navigate to the root folder.

Then, go File > Save Workspace As and save the workspace within the root folder.

This will create a workspace file, which will allow VSC to remember our code and help us debug.

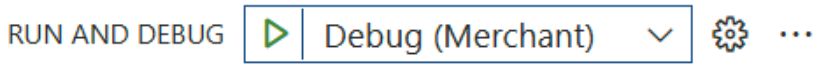
Next, go to the 'Run and Debug' menu, or press Ctrl + Shift + D to open it via a shortcut. You will see that VSC doesn't know how we should launch our python application. Let's fix that.

Create a folder called .vscode in our root folder, and inside create a folder called 'launch.json'. This file will tell VSC how to properly run our application.

Inside, add the following code:

```
{
  "version": "1",
  "configurations": [
    {
      "name": "Debug",
      "type": "debugpy",
      "request": "launch",
      "program": "${workspaceFolder}/Code/main.py",
      "cwd": "${workspaceFolder}"
    }
  ]
}
```

Now if you go back to VSC, you should see this pop up in the top right corner:



Hitting this will run our code and open up a terminal for it to run in. Try this now and ensure it works.

The benefits of doing this is that if something goes wrong in our code, our IDE will pause and take us to where the error has occurred. So we should use VSC to edit and test our game, while our run.bat is good for people to play our game once it is working.

2. Building the world

In MERCHANT, we want to define a number of ports, each with their own unique name, description and population.

Create a file in the code folder called map.py.

Within this file, we are going to define all the ports that the player can visit in their ship. Each port is going to be defined as a dictionary which contains the following information:

- Name - The name of the port
- Description - A brief description of the port, get creative here and add some cool details to your world
- YearFounded - The year the port was founded.
- Population - The population of the port
- Location - Where the port exists on the map, for simplicity, don't put your ports further than 25 away in any direction.

Here is a port that I have created:

```
Rye = {
    "Name" : "Rye",
    "Description" :
    """A vast town built on the Great Midland plains, Rye is irrigated by
    great canals made of shining white limestone and exports vast amounts
    of wheat, barley and apples. May come to visit the nearby orchands,
    enjoy the pleasant weather and sample the region's fine liquor.""",
    "YearFounded" : 441,
    "Population" : 25291,
    "Location" : (-4, 8)
}
```

TASK : Go ahead and create 6 - 12 ports for your world.

- Consider drawing a map on paper and figuring out where they will go.
- I'm styling my game as a fantasy-sailing game, but you can style it however you want! It could be set in space and the player has a space ship, or you could be riding trains across early 1800s America; You decide what theme and world you think is cool.
- Don't worry if you can't think up interesting descriptions yet, you can always come back and change it later once you've thought up a cool story for your world.

Once you have come up with all your ports, create a function called 'getAllPorts' which returns all the ports you have made as a list:

```
40  def getAllPorts():
41      # In my game, I have made 3 ports - Oxford,
42      # Nelson's Reach, and Rye. You should have
43      # Around 6 - 12
44      allPorts = [Oxford, NelsonsReach, Rye]
45      return allPorts
```

Let's head back to main.py now, and import these ports:

Write the following code and test that it works:

```
1   from map import getAllPorts
2
3   print ("Welcome to MERCHANT")
4
5   allPorts = getAllPorts()
6
7   print("### Ports:")
8   for port in allPorts:
9       print ("- " + port["Name"])
10
11  input("\nThe program has finished. Press any key to exit.
```

When you run your game with run.bat, it should list all the ports in the game.

3. Setting up the player

Let's create a new python file in the code folder called 'player.py' - This file will contain all the information regarding the player.

In player.py, set up the following Player class.

```
1   class Player:
2       # The name of the player's ship
3       shipName = "Unnamed Ship"
4
5       # The name of the port that the player is in
6       currentLocation = ""
7
8       # How much money the player has
9       money = 50
10
11      # How far the ship can sail
12      shipRange = 12
13
14      # How many tonnes of cargo the ship can hold
15      shipCargoSpace = 5
16
```

You can import this class into main.py the same way you imported the getAllPorts() function from the map class.

```
from player import Player
```

A player object can be created like so, and you can edit the player's data easily:

```
 9   #Setup player data
10   player = Player()
11   player.shipName = "HMS Victory"
```

TASK - When the player opens the game, they should be greeted with a message welcoming them to the game and then asked to enter their ship's name.

TASK - Add another function to map.py called **getStartingPort** which returns the port that the player should start in. Set the player's currentLocation to be this port's name.

- Don't forget to import this function too!

Once you have done these, we should be able to print out the following with no errors:

```
print("The valiant frigate " + player.shipName + " is currently docked at " + player.currentLocation + " harbour.")
```

4. Setting Sail

To calculate the distance between two ports, we can calculate the horizontal and vertical distances and then use Pythagoras theorem. Here is how you can manually calculate it: (Note that you'll need to import math)

```
# Ofxord is at position (-12, -2)
# Rye is at position (-4, 8)

oxfordX, oxfordY = Oxford["Location"]
ryeX, ryeY = Rye["Location"]

totalX = ryeX - oxfordX
totalY = ryeY - oxfordY

distance = math.sqrt(math.pow(totalX, 2) + math.pow(totalY, 2))
```

TASK - In map.py, create a function called findDistanceBetweenPorts which takes in two ports and returns the distance between them.

- Test this function by listing all ports and the distance to them from the currently docked port.

You might have noticed that we defined a 'shipRange' value in the player class. This limits how far players can travel. If you carefully design the map, you will be able to force players to frequently travel through some ports as they move around the map.

We can create the following function in map.py to get a list of ports and their distance to the current port:

```
62  def getAllPortDistancesFromPort(currentPort):
63      portDistances = []
64      # Loop over all the ports in the game:
65      for port in getAllPorts():
66          # If this port is the one we are at, skip over it
67          if (port["Name"] == currentPort["Name"]):
68              continue;
69          # Find the distance to this port
70          distance = getDistanceBetweenPorts(currentPort, port)
71          # Save it as a special 'portDistance' dictionary
72          portDistance = {
73              "Port" : port,
74              "Distance" : distance
75          }
76          portDistances.append(portDistance)
77
78      return portDistances
```

TASK - Expand the above function to return a sorted version of the portDistances list, sorting with closest ports first and the further away port last.

To make things easier for us, we can now define a function on the Player class that can tell us if the player is in range of a port - Just make sure to import the getDistanceBetweenPorts function from map.py, and indent this function inside of the player class:

```
19      def isPortWithinRange(self, desinationPort):
20          return getDistanceBetweenPorts(self.currentLocation, desinationPort) < self.shipRange
```

We can then use it like so:

```
18  # This will tell me if the port of Rye is within range:
19  if (player.isPortWithinRange(Rye)):
20      print("Rye is within range")
21  else:
22      print("Rye is not within range")
```

TASK - In main.py, create a function called getPortToTravelToUI. This function should list all ports in the game in order of their distance to the player. It should return the port that the player has chosen to sail to. If the port is further away than the player's ship can sail, put some text next to it saying it is too far away.

- This function should prompt players to enter the name of the port they wish to travel to.

- If the player enters a port that is too far away, they should be told it is too far and asked again
- If the player enters a port that doesn't exit, they should be asked to enter again.
- If the player types 'back', a value of -1 should be returned.

One final thing before finishing this section: We can make our game look a lot prettier if we use the os.system('cls') command, which clears the console. We just have to import os first.

```
18  def getPortToTravelToUI():
19      os.system('cls')
20      print("Which port shall we set sail for Capt'n?")
21      print("Or type 'back' to cancel\n")
22      # More code below...
```

To make our game have a better user interface, it's good practice to clear the console whenever the UI changes - for example when the player enters a new port.

5. Establishing the game loop

We can create a game loop by using a while loop to lock the thread until it receives proper input from the player.

First, create a function called currentPortUI which displays information about the current port:

```
46  def currentPortUI():
47      os.system("cls")
48      print("--- " + player.currentLocation["Name"] + " ---\n")
49      print("Established " + str(player.currentLocation["YearFounded"]) +
50            " | Population " + str(player.currentLocation["Population"]))
51      print("\n" + player.currentLocation["Description"])
52      input("\n(Press any key to return)")
```

Let's also create a small function called called sailingUI which indicates to the player that we are travelling between ports:

```
54  def sailingUI():
55      os.system("cls")
56      print("... Travelling ...")
57      input("...")
```

Now we can create our main gameplay loop in a mainUI function:

```
19  def mainUI():
20      while (True):
21          os.system('cls')
22          print("The valiant frigate " + player.shipName + " is currently "+
23              "docked at " + player.currentLocation["Name"] + " harbour.")
24
25          # Input info:
26          print("\n What are we t'do in port, capt'n?")
27          print("ashore - Vist the settlement we are currently docked at")
28          print("sail - View other ports to travel to")
29          print("quit - Quit the game")
30
31          # Take player input:
32          playerInput = input("> ")
33          if (playerInput == "quit"):
34              return;
35          elif (playerInput == "ashore"):
36              currentPortUI();
37          elif (playerInput == "sail"):
38              # Go to the screen where players can choose where
39              # to travel to:
40              portToTravelTo = getPortToTravelToUI();
41              if (portToTravelTo == -1):
42                  continue
43              player.currentLocation = portToTravelTo
44              sailingUI()
```

You'll notice again in these functions that we're calling os.system('cls') to clear the console. You'll also find that if unexpected player input is given, the while loop just loops back around to the same menu.

Test the game now and validate that everything works as intended.

6. Setting up the Hold

Now that we have our basic game loop working, and can travel between cities, it's time we started on the gameplay - Buying and selling!

Create a new file called cargo.py, in it define the following class - You will see that it is extremely similar to the way we are defining ports, except I've opted to use classes rather than dictionaries.

```
1   class Cargo:
2       name = "Unnamed cargo"
3       basePrice = 1
4
```

Now we can set up some cargo instances and define a getAllCargo function to give us all the cargo in the game. You should be able to see many similarities to the map.py file.

```
5   ale = Cargo()
6   ale.name = "Ale"
7   ale.basePrice = 12
8
9   blackPowder = Cargo()
10  blackPowder.name = "Black Powder"
11  blackPowder.basePrice = 18
12
13  wheat = Cargo()
14  wheat.name = "Wheat"
15  wheat.basePrice = 7
16
17  def getAllCargo():
18      allCargo = [ale, blackPowder, wheat]
19      return allCargo
20
```

TASK - Create 4 - 8 unique cargo types for your game. Make sure they fit in with the theme of your world.

TASK - Import the getAllCargo function into main.py, and loop over all the cargo types in the game to prove they are working correctly.

Once we have cargo in the game, we need to define a relationship between the cargo types and the cargo we have in the hold.

We've already defined our ship's cargo capacity, but now we are going to define:

- The total amount of cargo in the hold as an integer.
- The amount of each cargo in our hold as a dictionary.

Defining the amount of cargo is simple, we can just add a variable to our player class:

```
20      # How many tonnes of cargo is currently in the hold
21      # Cannot be higher than shipCargoSpace
22      shipCargoTotalAmount = 0
```

Storing the amount of each type of cargo is going to be a little more tricky. We'll use a dictionary for this, and a function to set the dictionary up:

```
24      # Dictionary relating cargo types to the amount of it in the hold:
25      shipCargoInventory = {}
26
27      def setupCargoInventory(self):
28          allCargo = getAllCargo()
29          for cargo in allCargo:
30              self.shipCargoInventory[cargo] = 0
```

You might be tempted to call this function in main.py, just after we set up the player, like so:

```
8   #Setup player data
9   player = Player()
10  player.setupCargoInventory()
11  player.currentLocation = getStartingPort()
```

However, since the player is a class, we can make use of *object oriented programming* to make life easier for us.

In python, all classes have a function called __init__() which is called when a class instance is created. So if we go back to the player class we can do the following:

```
35      def __init__(self):
36          self.setupCargoInventory()
```

Now, when we create our player in main.py with the line: 'player = Player()', the __init__ function gets called automatically. To test this is working for yourself, try the following code:

```
35      def __init__(self):
36          self.setupCargoInventory()
37          print("Player is initialised!")
38          input()
```

Object Orientation is so cool!

TASK - In main.py, create a method called holdUI which displays the state of the hold.

- The player should be able to access this holdUI from the mainUI like they can access the other UI panels.
- This screen should also show the capacity of the ship's hold.

Here is what it should look like once you've got it working:

```
--- HOLD ---

Capacity: 0 / 5 tonnes

0t - Ale
0t - Black Powder
0t - Wheat

(Press any key to return)
```

7. Building the market

Now that we have the ship's hold set up, it's time to set up the market. This is where we can buy and sell goods in the port.

Before we can set up the market, it's time for us to clear up some **technical debt**.

But what is **technical debt?**

Technical debt refers to the future costs associated with choosing quick or easy solutions in software development instead of more robust ones.

Our initial decision to store our ports as dictionaries made sense when we were trying to get our project up and running. But now that we want more functionality (namely, buying and selling goods at the market), switching to using a class is going to make our life a lot easier in the long run.

Let's head over to map.py and set up the following class:

```
41  class Port:
42      name = "Unnamed port"
43      description = "No description."
44      yearFounded = 0
45      population = 0
46      location = (0, 0)
```

TASK - *Refactor* our code to use this Port class rather than the dictionaries we have been using so far. This will require you to make changes in map.py and main.py.

OPTIONAL TASK: Loading ports from JSON

Keeping our ports stored in python's dictionary format is actually very useful. When designing a game, it is good practice to separate our content from our functionality.

In terms of our game, the content is our ports & cargo types, while our functionality is all the game logic.

We want to keep these separate so that our programmers can work on the functionality (adding more code and gameplay mechanics), and our designers & artists can work on our content (Adding more ports and cargo types).

To make our code super extendable, we can save our port data in **J**ava **S**cript **O**bject **N**otation (JSON).

JSON is a data format that we can use to define objects, originally created for the language JavaScript, it's now used just about everywhere.

Create a new folder in the root called 'Content' and create a new folder called ports.json in it. We can define ports like this:

```
[
    {
        "name" : "Oxford",
        "description" : "A quiet seaside port settled ir
        "yearFounded" : 621,
        "population" : 14402,
        "locationX" : -12,
        "locationY" : -2
    },

    {
        "name" : "Nelson's Reach",
        "description" : "A small port settlement founde
        "yearFounded" : 1012,
        "population" : 9021,
        "locationX" : 5,
        "locationY" : 7
    },

    {
        "name" : "Rye",
        "description" : "A vast town built on the Great
        "yearFounded" : 441,
        "population" : 25291,
        "locationX" : -4,
        "locationY" : -8
    }
]
```

You might notice that this format is **almost exactly the same** as the way we define dictionaries in python - cool huh?

First, import os and json to our map.py class:

```
import os
import json
```

Now that we've defined our ports in JSON, we need to read the file from our content folder. Inside of our getAllPorts function in map.py, we can use the following code to read our json file.

```
13  def loadAllPortsFromJson():
14      # Get our current directory, this will be our root folder.
15      currentDirectory = Path.cwd()
16
17      # Get the path to the ports.json file:
18      portsJsonPath = currentDirectory / "Content/ports.json"
19
20      # Try to read json
21      rawPorts = []
22      try:
23          with open(portsJsonPath, 'r') as portsJson:
24              return json.loads(portsJson.read())
25      except FileNotFoundError:
26          print("ports.json not found at directory " + str(portsJsonPath))
27          input("...")
28          return -1
29
```

The above function loads all the ports as a list.

OPTIONAL TASK - In the getAllPorts() function, call the loadAllPortsFromJson function, and then convert the list of json-loaded port objects into new instances of the port class. Here is some code to get you started:

```
31  def getAllPorts():
32      allPorts = []
33      allLoadedPorts = loadAllPortsFromJson()
34
35      if (allLoadedPorts == -1):
36          return []
37
38      for loadedPort in allLoadedPorts:
39          port = Port()
40
41          # Load the port data here:
42          # port.name = ...
43          # port.description = ...
44
45          # Add to the list
46          allPorts.append(port)
47      return allPorts
```

Remember to change the getAllPortDistancesFromCurrentPort function so that it doesn't call the getAllPorts function, as that now loads in all of our ports.

Create a new file in our code folder called market.py - This is where we'll store our data regarding the prices of the different goods that players can buy and sell.

We'll set up a market class that holds two dictionaries:

- One for the buying prices of goods,
- One for the selling prices of goods.

We also want to create a function called refreshPrices which is where our prices will eventually be randomised slightly, for now we'll just set the buy and sell prices to match the basePrice of each good.

```
1   from cargo import getAllCargo
2
3   class Market:
4       # The prices the player has to pay when buying goods:
5       buyPrices = {}
6       # The prices the player has to pay when selling goods:
7       sellPrices = {}
8
9       def refreshPrices(self):
10          allCargo = getAllCargo()
11
12          for cargo in allCargo:
13              self.buyPrices[cargo] = cargo.basePrice
14              self.sellPrices[cargo] = cargo.basePrice
15
```

TASK - Update the market class so that refreshPrices is called in the __init__ function.

Update the port class to have a variable called 'market'. A new instance of the market class should be created in the port class's __init__ method:

```
6   # Ports:
7   class Port:
8       name = "Unnamed port"
9       description = "No description."
10      yearFounded = 0
11      population = 0
12      location = (0, 0)
13      market = -1
14
15      def __init__(self):
16          self.market = Market()
```

Now, when we create our ports in the getAllPorts function, we will create new market instances (remember __init__ is called whenever we create a new class instance). When each market instance is created, it will then call its refreshPrices method

TASK - Setup a new method called marketUI. This should list all the cargo and its buy and sell price. It should be accessible from the mainUI. When you've got it working it should look something like this:

```
--- Oxford Market ---

Ale, Buy: £12, Sell: £12
Black Powder, Buy: £18, Sell: £18
Wheat, Buy: £7, Sell: £7

back - Leave the market
>
```

OPTIONAL TASK - While it's nice that we can see the buy and sell price of each good, it would look much prettier if it was aligned like a table. Install the prettytable package with the command 'pip install prettytable' in the terminal. Then use it to format the market menu so that it looks similar to the image below. Note that this is an exercise in reading documentation and independent research - go to https://pypi.org/project/prettytable/

```
--- Oxford Market ---

Name                Buy         Sell
Ale                 £12         £12
Black Powder        £18         £18
Wheat               £7          £7

back - Leave the market
> 
```

TASK - Add to our marketUI so that players can type ‘buy’ to move to a purchase screen. This screen needs to show them how much is in their hold, and how much free space they have, and how much money they have.

```
--- Oxford Market ---

Cargo Capacity: 0 / 5 tonnes
Money: £50

Name                Price/Tonne
Ale                 £12
Black Powder        £18
Wheat               £7

What would you like to purchase?
Or type 'back' to return
> 
```

In player.py, let’s add a method called attemptPurchaseGoods - This is where we will keep the logic that decides whether the player is able to buy goods and adding purchased cargo to their inventory.

```
38      # CargoToBuy - The type of cargo to purchase
39      # priceOfCargo - The cost of each piece of cargo
40      # Quantity - How much to buy
41      # Returns two variables:
42      #  > a boolean to indicate whether the purchase succeeded
43      #  > a string message to be given to the player
44      def attemptPurchaseGoods(self, cargoToBuy, priceOfCargo, quantity):
45          # Check there is enough room in the hold:
46          roomLeftInHold = self.shipCargoSpace - self.shipCargoTotalAmount
47          if (roomLeftInHold < quantity):
48              return (False, "Not enough room in the hold.")
49
50          # Check we have enough money
51          totalPrice = int(priceOfCargo * quantity)
52          if (totalPrice > self.money):
53              return (False, "Not enough money.")
54
55          # If we do have space in the hold and do
56          # have enought money, we are good to buy:
57          self.money -= totalPrice
58          self.shipCargoTotalAmount += quantity
59          self.shipCargoInventory[cargoToBuy] += quantity
60
61          return (True, "Success!")
```

TASK - Use the above function in player.py so that players can purchase goods from the market. Make sure you test your code to make sure that the player cannot buy more goods than they have money for, or room for in the hold. Once you have bought some goods, go check the hold to make sure that the goods are visible there.

```
--- Oxford Market ---

Cargo Capacity: 0 / 5 tonnes
Money: £50

Name                Price/Tonne
Ale                 £12
Black Powder        £18
Wheat               £7

What would you like to purchase?
Or type 'back' to return
> Ale
How much?
> 3
Success!
...
```

```
--- HOLD ---

Capacity: 3 / 5 tonnes

3t - Ale
0t - Black Powder
0t - Wheat

(Press any key to return)
```

TASK - Expand the game so that the player can sell goods they have in the hold. This should be very similar to buying goods. Make sure that the player can't sell more goods than they have.

- When listing goods to sell, you should only show the goods that are currently in the ship's hold.
- As a bonus task, make it so that the player can input 'all' as a quantity to sell all of their goods, rather than having to type the amount they have.

```
--- Oxford Market ---

Cargo Capacity: 3 / 5 tonnes
Money: £29

Name          In Hold        Price/Tonne
Wheat         3              £7

What would you like to sell?
Or type 'back' to return
> Wheat
How much?
> 3
Success!
...
```

8. Price Fluctuations

It's not time that we start working on getting some price fluctuations!

Currently, you can buy and sell goods at exactly the same price, which isn't very interesting. We're going to do 2 things in this section:

- Make it so that buying is normally more expensive than selling.
- Add in some random noise to the market, so that prices are a little bit random.

Let's get started over in market.py, and add in a general 20% increase to prices when selling. I'm going to create two functions: findSellPrice and findBuyPrice to help us separate our code a bit better:

```
12      def refreshPrices(self):
13          allCargo = getAllCargo()
14
15          for cargo in allCargo:
16              self.buyPrices[cargo] = self.findBuyPrice(cargo)
17              self.sellPrices[cargo] = self.findSellPrice(cargo)
18
19      def findSellPrice(self, cargo):
20          return cargo.basePrice
21
22      def findBuyPrice(self, cargo):
23          # Start by converting the basePrice to a float
24          # Remember that floats, unlike ints, have
25          # Decimal points.
26          price = float(cargo.basePrice)
27
28          # Make everything 20% more expensive
29          price = price * 1.2
30
31          # Convert price back to an int when we return it
32          return int(price)
33
```

Nice! That just made our game a whole lot harder, as now the player will lose money whenever they buy and sell normally, which is good, as we want the player to strategically buy and sell when prices are favourable.

TASK - Implement randomness to prices, so they can be randomly more expensive or cheaper.

- It's important to make sure that the price randomness is applied to both the sell and buy price of each good. Make sure you roll the price randomiser only once for each cargo type.

Here is how you can generate random numbers with Python:

```
import random

# This will give us a random number between 0.75 and 1.25:
priceRandomiser = random.uniform(0.75, 1.25)

# This will randomise the price, it will be a somewhere between 7.5 to 12.5
price = 10
randomisedPrice = price * priceRandomiser
```

When you've got this working, try to play the game and see if you can turn a profit! It might be harder than you expect. Try adjusting the price randomness as you play.

TASK - When we enter a port, call the market's refreshPrices function. This makes sure that the player never gets the same prices twice and when you leave a port and return the prices have changed.

9. Feast & Famine

To make the price fluctuations even more interesting, let's add in random shortages and surpluses to goods in certain markets. Let's say that there is a small chance that a good can be either in a **surplus** or in a **shortage**. With each increasing or decreasing prices by 25%.

Here is a function that will have a 10% chance of shortages or surpluses:

```
51      def rollForMarketEvent():
52          # 10% chance of rolling a surplus
53          randomRoll = random.uniform(0, 1)
54          if (randomRoll < 0.1):
55              return "Shortage"
56          # 10% chance of rolling a surplus
57          randomRoll = random.uniform(0, 1)
58          if (randomRoll < 0.1):
59              return "Surplus"
60
61          # Else return null
62          return None
```

TASK - Implement surpluses and shortages into markets, so that there is a small chance of their happening when a port refreshes its prices. Make sure that the marketUI now reflects this, so when a player views the prices it might show this:

```
--- Oxford Market ---

Cargo Capacity: 0 / 5 tonnes
Money: £50

Name                Buy         Sell
Ale                 £15         £12
Black Powder        £18         £15
Wheat               £5          £4          Surplus

back - Leave the market
buy - Purchase some goods
sell - Sell some goods
```

10. Next Steps

That's it. We've now finished off the basic framework for our game. Here is where you get to improve it and make it your own.

Optional Task - Travelling Costs

To make our game more difficult, and more strategic, it would be nice if the player had to pay their crew every time they sailed.

- This means the player can now lose the game, by going fully bankrupt.
- You might want to charge a flat fee for every time the player sets sail, plus an additional fee for the distance travelled. This will make longer voyages more difficult.

Optional Task - You've got mail!

To make the game a little bit easier, you might want to give the player opportunities to deliver mail from each port. They should be able to visit a post office in each port which should randomly offer some letters to be delivered to nearby ports, paying more the further the player has to travel.

- This will allow the player to make some small income just from moving around, which could offset the travelling costs from the previous optional task.

Optional Task - Upgrading your ship

Now we've got all this money from buying and selling, we need to figure out what to do with all of it. Implementing upgrades to our ship would be fantastic, and would add a sense of progression to our game.

- Expanding the hold and increasing the ships range would be a good place to start.
- This optional task works really well with the pirates optional task, as you might want to be able to upgrade the ship's cannons and armour to better defeat pirates.

Optional Task - Voyage encounters

To make sailing more interesting, try writing 10 or so random encounters that have a chance to occur when sailing between ports. This gives you a chance to do some interesting creative writing.

- These random encounters should have some element of player choice and randomness. For example, the player ship might encounter a treasure chest floating in the sea, they can choose whether to open it or leave it alone. Opening it should give the player a random bit of cargo, or have some other random effect happen.
- This can synergise really well with the pirates optional task.

Optional Task - Pirates!

We've come this far, why not add pirates to our game.

- This is your chance to try to create an interesting combat mechanic in our game.
- Try to think how you could add an element of strategy to this. Perhaps the player can decide between flanking, charging or retreating, and each having a different result.
- You don't have any cool graphics here, and everything has to be turn based. Embrace these restrictions.

Optional Task - Improved mao

Depending on how you've set up your map, you might have found that some of the port distances don't make much sense. For example, if you have an island, and two ports on either side of the island, the player's ship will be able to sail directly from one port to the other, as though it was an airplane rather than a ship.

- To fix this, look at creating a dictionary that keeps track of the distances between ports, this will allow you to define how long it takes to travel between ports, rather than the existing distance calculation.

- To make it even better, look at creating a map using ASCII art, you could even draw a special character on the map to represent the player and each port.