ICS 226: Network Programming

Lab Manual

Fall 2023

Michael Horie
Camosun College

Lab 1: Set Up the ICS 226 Dev Environment

Please see the course outline for the due date

Preparation

- ☐ If you haven't done so already, pick a free workstation that belongs to your section, and provide the number to your instructor
- ☐ Obtain the following from your instructor:
  - ☐ your Ubuntu VM IP address, user ID, and password
  - ☐ the SQP URL, user ID, and password

A. Test Access to SQP

- ☐ Log in to your A machine
- ☐ Using a web browser, go to the SQP URL. If you have never used SQP before, this website is used to page your instructor when you would like to demonstrate a milestone or have any questions. It ensures that demos are viewed and questions are answered in priority order (demos first, then support questions), as well as to keep response times within reasonable limits. From now on, please use SQP to contact your instructor during each lab. Use the *Demo* request to indicate that you want to demonstrate a milestone, and use the *Support* request to ask a question (please provide a brief summary of your question; this

allows for quicker responses). Also, while waiting, feel free to continue with the lab until your instructor responds to your request

- ☐ Enter your SQP user ID and password
- ☐ Make a *Demo* request.

SQP Works /1
(Instructor Stamp)

B. Connect the Ubuntu App to your VM

- ☐ Click on the Windows *Start* menu and type in *ubuntu*
- ☐ If the Ubuntu app shows up, run it. If not, download the Ubuntu app from the Microsoft Store, making sure that it is the one from Canonical Group Ltd.
- ☐ In the Ubuntu app, enter the command ssh-keygen and keep pressing the *Enter* key to go with the default values. This creates a new SSH key on your A machine
- ☐ Enter the command ssh-copy-id _____@_____ where the first blank is the VM user ID and the second blank is the VM IP address given to you by your instructor. This starts a process that will allow you to log into your VM using only the SSH key you just generated
- ☐ Accept the fingerprint
- ☐ Enter the VM password given to you by your instructor
- ☐ Now enter the command ssh _____@_____ where the first blank is the VM user ID and the second blank is the VM IP address, to log into your VM; make sure that you can log in without entering a password, or something went wrong
- ☐ Enter the command exit to leave the remote shell
- ☐ Enter the command explorer.exe . (don't forget the dot at the end!) to reveal the current directory in an Explorer window
- ☐ Leave the Ubuntu and Explorer windows open

C. Connect PyCharm to your VM

- ☐ Click on the Start menu and type in *pycharm*
- ☐ Start PyCharm
- ☐ Under *Remote Development > SSH*, click on *New Project*
- ☐ For the *Username* and *Host*, use the VM user ID and IP address you used above
- ☐ Enable *Specify Private Key*
- ☐ Double-click on the folder icon in the text field that just appeared
- ☐ Drag the *.ssh* directory from the Explorer window into the file panel and wait until the search completes
- ☐ In PyCharm, open the *.ssh* directory that you dragged into the panel and select *id_rsa*
- ☐ Click on *OK*
- ☐ Click on *Check Connection and Continue*
- ☐ Click on *Open an SSH terminal*
- ☐ We must now create a project directory. In the terminal that appears, enter the command mkdir 226_lab0

- [ ] Enter the command exit
- [ ] In PyCharm, select *226_lab0* as the *Project Directory*
- [ ] Click on *Download IDE and Connect*
- [ ] **After** the setup completes, you should see a file called *main.py*. If not, click on *Remote Development > SSH* and click on the *226_lab0* link. PyCharm is now set up to allow you to view, edit, run, and debug code on your VM from your A machine

D. Install the Latest Stable Python Interpreter

- [ ] From within PyCharm, click on the *Terminal* icon in the left sidebar to open a shell on your VM, with the working directory set to the project directory
- [ ] Enter the command sudo apt-get update
- [ ] Enter the command sudo apt-get dist-upgrade
- [ ] Install all suggested packages, if there are any
- [ ] Restart all services, if so prompted
- [ ] Now that all existing software on your VM is up to date, install the latest, stable version of Python (at time of printing) using the command sudo apt-get install python3.11 python3-pip
- [ ] Install *pipenv* using the command pip install --user pipenv
- [ ] Click on *Python 3.10* on the bottom status bar
- [ ] Click on *Add New Interpreter > Add Local Interpreter...* (you may have to click on the > for the popup menu to show up)
- [ ] Select ***Pipenv*** *Environment*
- [ ] If required, set the *Base Interpreter* to */usr/bin/python3.11*
- [ ] If the *Pipenv Executable* path is empty:
  - [ ] click on the folder icon,
  - [ ] enable the display of hidden files and directories, and
  - [ ] select *pipenv* in the *.local/bin* directory inside your VM's home directory
- [ ] Click on *OK*
- [ ] If the Interpreter window pops up again, just cancel it and wait for the background tasks listed in the bottom status menu to finish
- [ ] Click on the *Debug* icon in the top menubar to run the code
- [ ] Confirm that *Hi, PyCharm* appears in the console at the bottom of the window

E. Connect Your Project to GitHub

- [ ] Create a GitHub account using your generic Camosun email address (the one starting with the letters *ics*). If you don't recall that email address, please check your *M* drive for details. It is recommended that you don't use your personal GitHub account
- [ ] From the PyCharm Terminal you used in the above section, enter the command ssh-keygen and keep pressing the *Enter* key to go with the default values (while you already created an SSH key on your A machine, your VM does not yet have one)

- [ ] Enter the command `cat ~/.ssh/id_rsa.pub` (don't forget the 2 dots in that command!) to print out the public portion of the SSH key (you will need this SSH key shortly)
- [ ] Using a web browser, go to https://github.com
- [ ] Go to *Settings > SSH and GPG keys*
- [ ] Click on *New SSH Key*
- [ ] For the title, enter *ICS226VM*
- [ ] For the key, copy and paste the SSH key you printed out above
- [ ] Create a new **private** repository called *226_lab0*
- [ ] Go back to the Terminal
- [ ] Confirm that you are in the *~/226_lab0* directory
- [ ] Now create a git repository and link it to GitHub by entering the commands:
  - [ ] `git init`
  - [ ] `git branch -m main`
  - [ ] `git config --global user.email "____"` where the blank is the email address you used to sign up with GitHub
  - [ ] `git config --global user.name "____"` where the blank is your GitHub user name
  - [ ] `git add .` (don't forget the dot at the end)
  - [ ] `git commit -m "Initial Commit"`
  - [ ] `git remote add origin git@github.com:____/226_lab0.git` where the blank is your GitHub user name.
  - [ ] `git push -u origin main`
- [ ] Confirm that GitHub hosts your *226_lab0* repository
- [ ] Create the work branch and switch to it by issuing the commands:
  - [ ] `git branch work`
  - [ ] `git checkout work`
  - [ ] `git push -u origin work`
- [ ] Add *print("Done")* at the end of *main.py*
- [ ] Click on the Version Control icon in the left sidebar
- [ ] Confirm that the change you just made is highlighted
- [ ] Right-click on *main.py* and select *Rollback*
- [ ] Click on *Rollback*
- [ ] Note that the change has been undone. You have restored everything to the last commit point. While undo (Ctrl + Z) takes you back one step at a time, rollback allows you to undo all the way to the last commit point in one go
- [ ] Show your instructor that your repository on GitHub is private

Repo is Private /3
(Instructor Stamp)

F. Prepare for the Game Project

- [ ] Based on the instructions in this lab:

- ☐ Using the Ubuntu App, log into your VM
- ☐ Create a new directory called *226_game* in your **home** directory (**not** in the *226_lab0* directory)
- ☐ Create a new PyCharm project using the **existing** PyCharm connection to your VM, and *226_game* as the *Project Directory*
- ☐ Change the source code to print out the greeting *Hello World*
- ☐ Make sure that you can run the program in PyCharm; you may have to re-select */usr/bin/python3.11* as the interpreter
- ☐ Download the *.gitignore* file from D2L and save it into your game directory. You can use WinSCP to copy files between your A machine and your VM
- ☐ Create a new **private** repository on GitHub called *226_game*
- ☐ From within the *226_game* directory, create a new git repository and connect it to your GitHub *226_game* repository (you don't have to repeat the *git config* commands; they only have to be issued once)
- ☐ Create a work branch and switch to it
- ☐ Commit the source code to GitHub

☐ Show the contents of your VM's **home** directory, the running program, and the GitHub repository to your instructor.

Completion

- ☐ Log out of your VM and your current machine
- ☐ Congratulations! You have completed this lab. See you next week!

VM Dir, Run, Private Work Branch /6
(Instructor Stamp)

Important Take-Aways

- ☐ Use SQP to contact your instructor during a lab
- ☐ *apt-get install* installs an Ubuntu package system-wide, *pip install --user* installs a Python package for the current user only; this reduces the chance of conflicts since the system also depends on Python packages
- ☐ *git init* creates a new Git repository inside the current directory
- ☐ *git branch* create a new Git branch; *git branch -m* renames the current branch
- ☐ *git config --global* changes a global Git configuration (only needs to be done once per user account)
- ☐ *git add* adds a file to the repository
- ☐ *git commit* saves a change to the local repository
- ☐ *git remote add* connects a local repository to the remote repository (only needs to be done once per repository)
- ☐ *git push* pushes a commit to a remote repository
- ☐ *git checkout* switches between branches
- ☐ *git merge* merges the changes of another branch to the current branch

Lab 2: Create the Game Model Logic

Please see the course outline for the due date

A. Create the Game Board Logic

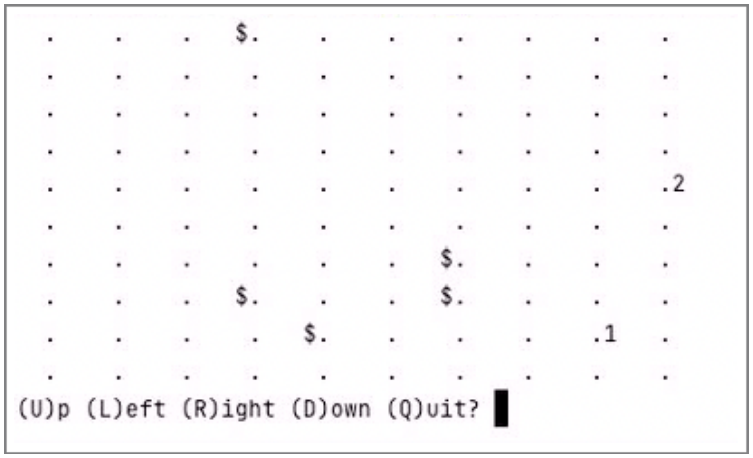
- ☐ Open your PyCharm *226_game* project
- ☐ From the PyCharm *File* menu, create new Python files called *Treasure.py*, *Tile.py*, *Board.py*, and *View.py*. Be sure to add them to Git when prompted to do so
- ☐ In *Treasure.py*, create a class called *Treasure* that contains a description string and a value integer instance variable. While the default description is '$', there is no default value for the integer
- ☐ In *Tile.py*, create a class called *Tile* that contains a description string and a *Treasure* instance variable. The default description is '.' and the default *Treasure* is *None*
- ☐ In *Board.py*, create a class called *Board* that creates a square *n x n* collection of *Tiles* with *t Treasures* randomly placed in it. Each *Treasure* is given a positive random value between *min_val* and *max_val*. The variables *n*, *t*, *min_val*, and *max_val* must be passed into the *Board* *__init__* method. Any invalid argument results in a *ValueError* exception; your task now is to find and check for all possible exceptions!
- ☐ In *View.py*, create a method called *display* that takes a *Board* and prints out that *Board* as a 2D structure
- ☐ Change *main.py* so that it creates a 10 x 10 *Board* with 5 *Treasures,* ranging from $5 to $10, and prints it out
- ☐ Run the program 10 times and confirm that 5 *Treasures* are randomly assigned to a 10 x 10 *Board* each time
- ☐ Add comments to all methods and functions, except *__str__*; note that PyCharm will create a comment template for you if you enter 3 quotation marks (""") in a new line right underneath a *def*, and then press the *Enter* key
- ☐ Confirm that the program still runs
- ☐ Commit and then push your changes to GitHub. You can use the PyCharm *Git* menu to do this. Be sure to provide a concise description of the changes in the comment field
- ☐ Confirm that the latest code is on GitHub

- [ ] Show the working program to your instructor

Board Generated /4
(Instructor Stamp)

B. Add the Player Logic

- [ ] In a new Python file called *Player.py*, add a *Player* class that contains a name string
- [ ] In the *Tile* class, add a method called *add_player* that adds a given player to the *Tile*; no error checking is required
- [ ] In the *Board* class
  - [ ] modify the *__init__* so that it also takes a maximum number of *Players*; make sure this is in the range of *0 <= max_players <= n*
  - [ ] add a method called *add_player* that takes a *name*, an *x*, and a *y*, and then creates a *Player* object with the given name and placed at the given position on the *Board. A ValueError* must be raised if that is not possible (e.g., out of range, already occupied)
  - [ ] add a method called *move_player* that takes a *name* and a *direction* (*up*, *down*, *left*, *right*) and attempts to move the Player with the given name in that direction; a *ValueError* is raised on error. Wrapping around (e.g., from the top row to the bottom row) is not allowed
- [ ] In *main.py*
  - [ ] add two *Players*, called *1* and *2*, to a free, random position on the board
  - [ ] repeatedly prompt the user for one of the following commands: *U* (up), *L* (left), *R* (right), *D* (down), followed by the Player number (1 or 2)
  - [ ] on error, print out the error message, but continue prompting
  - [ ] also add support for a *Q* (quit) command
- [ ] Don't forget to add (and update) comments!
- [ ] Commit and then push your changes to GitHub. Make sure to add a concise description of the changes
- [ ] Confirm that the latest code is on GitHub
- [ ] Show the working program to your instructor



SAMPLE SCREENSHOT

2 Players Supported /4
(Instructor Stamp)

C. Add the Score Logic

- [ ] Modify the code so that:
  - [ ] a *Treasure* is picked up by a *Player* when the *Player* reaches a *Tile* with the *Treasure*; the value of the *Treasure* is furthermore shown on pickup

- [ ] the game ends when there are no *Treasures* left
- [ ] each player's score, derived from the values of picked-up *Treasures*, is displayed on exit

- [ ] Don't forget to add (and update) comments!
- [ ] Commit and push your changes to GitHub. As before, add a concise description of the changes
- [ ] Confirm that the latest code is on GitHub
- [ ] Show the working program to your instructor

Scores Work /2
(Instructor Stamp)

Completion

- [ ] Log out of your VM and your current machine
- [ ] Congratulations! You have completed this lab. See you next week!

Important Take-Aways

- [ ] Be able to write programs using Python

Lab 3: Set Up the Test Environment

Please see the course outline for the due date

A. Install pytest

- ☐ Open your PyCharm *226_game* project
- ☐ Click on the Terminal tool in PyCharm (do **not** use the Ubuntu app for this section)
- ☐ Confirm that you are in the *226_game* directory
- ☐ Enter the command `pip3 install pipenv`
- ☐ If so prompted during the install, issue the command `pip install --upgrade pip`
- ☐ Enter the command `pipenv install pytest`
- ☐ Update everything using `pipenv update`
- ☐ Check for known security vulnerabilities in the installed libraries using the command `pipenv check`
  Contact your instructor if there are any failures

B. Add Tests

- ☐ Using PyCharm, create a new Python file called *test_game.py*
- ☐ Add the following code:

```
from Treasure import Treasure


def test_treasure():
    t1 = Treasure(10)
```

```
    t2 = Treasure(20, description='%')

    assert t1.value == 10
    assert t1.description == '$'
    assert t2.value == 20
    assert t2.description == '%'
```

- ☐ The above code checks if the *Treasure* initializer works correctly by first instantiating a *Treasure*, and then making sure that the instance variables were set correctly
- ☐ In the PyCharm Terminal, issue the command `pytest`
- ☐ Confirm that the test passed
- ☐ Add the following code at the bottom of *test_game.py*:

```
def test_board():
    with pytest.raises(ValueError, match='n must not be less than 2'):
        b = Board(1,5,5,10,2)
```

- ☐ You may have to update the exact error message to conform to your code
- ☐ Don't forget to import the *Board*
- ☐ Run *pytest* again to make sure that the tests pass
- ☐ Now add tests to check your game code **thoroughly**; you should have a test for every exception and for every method and for normal game play (except the *Board.__str__* and the *View* methods). The sample solution for this step consists of about 200 lines of code with **over 60** *assert* and *raises* checks!
- ☐ Commit and push your changes to GitHub
- ☐ Confirm that the latest code is now on GitHub
- ☐ Show the source code and test result to your instructor

Source Shown and Tests Pass /5
(Instructor Stamp)

C. Create a GitHub Action

- ☐ Log into GItHub. Under settings, please check your usage and make sure that you have enough free minutes left (you probably need around 10 for this lab; please keep an eye on your usage for the remainder of the course). Do not provide a credit card; if you have provided a credit card, please remove it before proceeding, to avoid charges
- ☐ Go to the *226_game* repository
- ☐ Click on *Actions*
- ☐ Click on *Configure* under *Python application*
- ☐ In the source code editor that appeared, change:
  - ☐ *Python application* to *Test Python Game*
  - ☐ Change *3.10* to *3.11* (2 places)
- ☐ Click on *Start commit*
- ☐ Click on *Commit new file*
- ☐ Click on the *Actions* tab again

- ☐ Click on the *Create python-app.yml* link
- ☐ Click on *build*
- ☐ The test should fail; this is because our source code and test code are still in the work branch, not the main branch
- ☐ Using the Terminal in PyCharm, enter the commands:
  - ☐ git checkout main
  - ☐ git pull
  - ☐ git merge work
  - ☐ git push
  - ☐ git checkout work
- ☐ Click on the *Actions* tab in GitHub again
- ☐ Click on the *Merge branch 'work'* link
- ☐ Click on *build*
- ☐ Expand *Test with pytest*
- ☐ Confirm that all tests passed
- ☐ From now on, whenever you push code onto the main branch on GitHub, the code will be tested; if the tests fail, GitHub will notify you by email and mark the commit as having failed the test

D. Refactor the Code

- ☐ Modify your game code so that no object directly accesses the instance variables of another object; for example, *Board* should not update the *Player* score directly, but should call an *add_score* method supplied by the *Player* class
- ☐ Update all comments
- ☐ Make sure the existing tests still pass
- ☐ Add tests to test the new methods you added
- ☐ Once the tests pass locally, push everything to your main branch, using the instructions in *Section C*, and confirm that the tests also pass on GitHub
- ☐ Show the result to your instructor

GitHub Tests Pass /5
(Instructor Stamp)

Completion

- ☐ Log out of your VM and your current machine
- ☐ Congratulations! You have completed this lab. See you next week!

Important Take-Aways

- ☐ *pipenv install* installs a Python package in a way that is local to a project directory; this allows different projects to use different packages and versions without causing a conflict (this is in contrast to *pip install --user* and *apt install* we used in Lab 1)

- ☐ The PyCharm Terminal automatically sets up a pipenv shell that ensures that any pipenv install command will install packages locally; the Ubuntu app does not do this automatically
- ☐ *pytest* is used to test Python code; it does this by looking for Python files starting with the word *test_*, and running functions starting with the word *test_* inside those files
- ☐ *pipenv check* is used to check for known vulnerabilities in imported Python packages
- ☐ All tests should set up a scenario and then verify the outcome using *assert* and *raises* calls
- ☐ The instruction sequence
  git checkout main
  git pull
  git merge work
  git push
  git checkout work
  merges changes from the work branch into the current main branch, pushes the result to GitHub, and reactivates the work branch. You must be in the correct project directory for these commands

Lab 4: Create a Networked Game Server

Please see the course outline for the due date

A. Create a TCP Server

- ☐ Make sure you are working on your *work* branch, **not** the *main* branch. You can use `git status` for that
- ☐ Modify your game code's *main.py* so that it is based on the *Sample TCP Server* slide, and:
  - ☐ accepts local and non-local TCP connections on port 12345,
  - ☐ receives a 1-byte segment *s* from a client where the first four bits indicate the command (*U* is 0010, *L* is 0100, *R* is 0110, *D* is 0011, *Q* is 1000, *G* --- get board --- is 1111), the next two bits indicates the Player number (Player 1 is 01 and Player 2 is 10; if the command is *Q* or *G*, this field is not relevant), and the next 2 bits are 00; an invalid command causes the connection to be closed immediately
  - ☐ prints out the IP address of the client, as well as *s* in binary form
  - ☐ if the command was *Q*, or the game is over, all subsequent commands are ignored and the connection is closed after transmitting the scores and game board
  - ☐ otherwise, implements the command
  - ☐ replies to the client with *a* segment *t* whose first packed unsigned short is the Player 1 score, and whose second packed unsigned short is the Player 2 score
  - ☐ if the command was *G*, furthermore sends the whole board as a UTF-8 encoded string
  - ☐ closes the connection
  - ☐ places all code into a class called *Game*; the only code outside of that class should be:

```
g = Game()
g.start()
```

- ☐ You can test your server using echo commands. For example:
  `echo -n 'F0' | xxd -r -p | nc 127.0.0.1 12345 | xxd`
  creates the string *F0* (without a newline), converts the hex string into plain bytes format, and transmits it to localhost port 12345, and prints out the response from the server (in this case, it should be the two scores and the board). Similarly,
  `echo -n '34' | xxd -r -p | nc 127.0.0.1 12345 | xxd`
  causes Player 1 to move down 1 position (make sure you understand how to derive those values; if in doubt, write out the command -- 0011 in this case -- and player number -- 01 in this case -- and 00 in binary, and convert that to hexadecimal)
- ☐ Once you are satisfied that your program is working correctly, run the tests from the last lab and make sure they all still pass
- ☐ Commit your changes and push them to GitHub; don't merge with the main branch yet, however
- ☐ Show your instructor that you can move both players on the board, and that scores are correctly updated

B. Observe Network Traffic

Game Server Works /4
(Instructor Stamp)

- ☐ Start your game server
- ☐ Start up Wireshark on your Windows machine
- ☐ Start recording your **Ethernet** (not loopback) traffic
- ☐ Filter using the expression
  `tcp.port == 12345`
  Don't forget to click on the blue arrow to apply the filtering expression
- ☐ From your Windows machine's **Ubuntu App**, enter the command
  `echo -n 'F0' | xxd -r -p | nc __.__.__.__ 12345 | xxd`
  where the blanks correspond to the IP address of your Ubuntu VM
- ☐ Stop recording your Ethernet traffic
- ☐ Using the Wireshark data, fill out the table on the next page. Please use a PDF editor, or take a screenshot of the table and edit that file; do not create a text or word processor document. For the *Phase* column, write
  - ☐ *UP* if the segment is part of the 3-way handshake,
  - ☐ *TX* if it is part of data transmission, and
  - ☐ *DN* if it is part of the connection teardown
- ☐ Verify that the data you just entered matches the TCP protocol discussed in class. If not, you most likely made a data entry mistake that needs to be corrected
- ☐ Submit the table to D2L.

Network Observations /2

No.	Source	Destination	Flags	Seq No.	Ack No.	Phase

C. Test the TCP Server Locally

- ☐ Make sure you are working on your *work* branch, **not** the *main* branch
- ☐ Download *test_server.py* and *Dockerfile* from D2L and save them in your game directory
- ☐ Install the docker.io package in your virtual machine; do not use *snap*, use *apt-get*; don't forget to update first!
- ☐ For purposes of thoroughness, the tester repeats its network test 10 times; however, for initial debugging purposes, this takes too long; so, change the *@pytest.mark.parametrize('execution_number', range(10))* to *@pytest.mark.parametrize('execution_number', range(1))* in *test_server.py*
- ☐ Run the command
  `pytest -s -v test_server.py`
  Note that this produces **lots** of output, which is helpful when debugging, but distracting when only looking for a summary. If you leave out the -s and -v flags, you can reduce the amount of output
- ☐ If you get a *Command '['nc', '127.0.0.1', '12345']' returned non-zero exit status* message, run
  `sudo docker run --rm --name 226-server -p 12345:12345 226-server`
  to get more information

- ☐ Fix your server code to pass the tests. For debugging purposes, connect to your VM from your Ubuntu app and execute the command
  `sudo journalctl -f`
  to observe the logs for potentially-helpful error messages
- ☐ Once your server passes, change the *@pytest.mark.parametrize('execution_number', range(1))* back to *@pytest.mark.parametrize('execution_number', range(10))*
- ☐ Make sure all tests still pass
- ☐ Show the result to your instructor

Local Tests Work /2
(Instructor Stamp)

D. Observe GitHub Actions

- ☐ Merge your changes with the *main* branch (if you don't remember how to do that, review the previous lab)
- ☐ Log into GitHub
- ☐ Go to the *226_Lab1_2022* repository
- ☐ Click on *Actions*
- ☐ Click on the top workflow
- ☐ Click on *build* (either one will work)
- ☐ Observe your tests in action. There are 16 tests (6 testing the game model, 10 testing the network). The percentage to the right will say *37%*, then *100%*. This is a progress indicator, not a correctness indicator. If the tests succeeded, the log should say *16 passed* at the end
- ☐ Show the result on GitHub to your instructor

GitHub Network Tests Work /2
(Instructor Stamp)

Completion

- ☐ Log out of your VM and your current machine
- ☐ Congratulations! You have completed this lab. **Be sure to hand in the Network Observations table** and see you next week!

Important Take-Aways

- ☐ *xxd* allow us to generate and view binary data from the command line
- ☐ Wireshark allows us to observe and troubleshoot the network aspect of an application
- ☐ Be able to write networking code in Python

Lab 5: Add Multithreading to the Server

Please see the course outline for the due date

A. Write a Game Client

- ☐ Make sure you are working on your *work* branch, **not** the *main* branch
- ☐ Add a file called *client.py* to your game project
- ☐ The client must:
  - ☐ initially contact the server to get a player ID, or to quit with an error message if 2 players are already registered
  - ☐ keep the connection to the server open until the client quits (if a network error causes the connection to fail, an error must be printed, but the connection should not be re-established)
  - ☐ repeatedly prompt the user for a direction command, or the quit command
  - ☐ convey the command to the server
  - ☐ show the latest board and scores after every direction command

  Also update the server to:
  - ☐ maintain a thread per connection
  - ☐ support the above client interactions
  - ☐ enforce mutual exclusion when accessing shared variables;
  - ☐ avoid locking while receiving and sending messages over the network
  - ☐ preface every network transmission with a size header (e.g., when sending the player scores, first send a 4, then the player scores, to indicate that the next transmission takes up 4 data bytes). The size header should be a single unsigned short
- ☐ Download the latest version of *test_server.py*
- ☐ Make sure all tests pass; update your server as required
- ☐ Merge everything with the main branch
- ☐ Push the changes to GitHub
- ☐ Show the working tests on GitHub to your instructor
- ☐ Using two clients, demonstrate your game interactively to your instructor

GitHub OK/Game Works /0
(Instructor Stamp)

Source Code
/10

B. Submit the Source Code

- ☐ Clean up your code to comply with Appendix A. **Take the time to write good comments for every function and instance variable!**
- ☐ Submit all your **Python** files to D2L as a zip file; do **not** use any other compression format. It is recommended to use *Code > Download ZIP* on GitHub. **Non-zip submissions, and submissions that cannot be unzipped on the default ICS 226 Ubuntu VM, will get a 0**

Completion

- ☐ Log out of your VM and your current machine
- ☐ Congratulations! You have completed this lab. **Be sure to hand in your source code** and see you next week!

Important Take-Aways

- ☐ Be able to write multithreaded networking code in Python (incl. prevention of race conditions)

Lab 6: Create an Asynchronous Server

Please see the course outline for the due date

A. Rewrite the Game Server Using asyncio

- ☐ Connect to your virtual machine
- ☐ Modify the **server** code so that *asyncio* is used instead of the socket and threading code used to date. Ideally, this should only require you to change the file containing the server code, not the file containing the game code
- ☐ Once everything is working, push the modified server code to your work and main branches on GitHub
- ☐ Show the working test to your instructor

asyncio Server Works /0
(Instructor Stamp)

B. Submit the Source Code

- ☐ Clean up your code to comply with Appendix A
- ☐ Push the changes to your work and main branches on GitHub
- ☐ Make sure that no tests fail
- ☐ Submit all your **Python** files to D2L as a zip file; do **not** use any other compression format. It is recommended to use *Code > Download ZIP* on GitHub. **Non-zip submissions, and submissions that cannot be unzipped on the default ICS 226 Ubuntu VM, will get a 0**

Source Code
/10

Completion

- ☐ Log out of your VM and your current machine
- ☐ Congratulations! You have completed this lab. **Be sure to hand in your source code** and see you next week!

Important Take-Aways

- ☐ Be able to write networking code in Python using asyncio

Lab 7: Use Django to Create a Server

Please see the course outline for the due date

Common Errors Encountered During Django Labs

- ☐ If python cannot find *manage.py*, make sure you are in the correct directory
- ☐ If python complains about a syntax error in *manage.py*, make sure you issued the command from within PyCharm or *pipenv shell*
- ☐ If you cannot connect to your Django website, make sure you are running *./manage.py runserver* and confirm that you have an ssh tunnel from your A machine to your VM
- ☐ If you get a *Page not found (404)* error, make sure you are using the correct URL; forgetting *game/* as part of the URL is a common error
- ☐ If you get an error message stating that the response does not have a *get* method, make sure that your view methods are returning answers wrapped in an *HttpResponse* class
- ☐ If a test does not run, make sure the method name starts with the word *test*
- ☐ If a test fails and you cannot determine why, make sure there is a */* at the end of every URL; if that still does not help, print out the response using *print(vars(response))* to give you a better idea of what is happening

A. Set up Django

- ☐ Before starting this lab, review the common errors; if you run into problems during this lab, review the common errors again before calling for assistance
- ☐ Confirm that you are on the *work* branch
- ☐ In PyCharm, install Django locally with

`pipenv install django`

- [ ] Just in case, update everything using
`pipenv update`
- [ ] Now create a default Django website using the command
`django-admin startproject website`
- [ ] Push what you have so far, including the new files, to your GitHub work branch
- [ ] Change into the *website* directory
- [ ] Run
`./manage.py makemigrations`
inside the website directory, followed by
`./manage.py migrate`
- [ ] Now run
`./manage.py runserver`
- [ ] Using either Powershell (Windows) or Terminal (macOS) on your machine, set up an ssh tunnel to your VM using the command
`ssh -L 8000:127.0.0.1:8000 student@__.__.__.__`
where the blanks indicate the IP address of your VM
- [ ] Open a web browser and go to http://localhost:8000
- [ ] Make sure you see the default Django web page.
- [ ] Using another PyCharm Terminal session, go to the *website* directory that contains *manage.py*, and create superuser credentials using the command
`./manage.py createsuperuser`
- [ ] Log in to http://localhost:8000/admin with the credentials you just created
- [ ] Make sure you can see the Django administration page
- [ ] Type
`git status`
You should see a file called *db.sqlite3* in the *~/226_game/website* directory (if it is in another directory, contact your instructor; do not proceed beyond this step before this error is corrected)
- [ ] We do not want to push the database to GitHub, since this file contains our admin account credentials (there are other security issues that we will look at when we deploy this app later on)
- [ ] In the *226_lab* directory, add the following to *.gitignore*
`db.sqlite3`
- [ ] Save the file
- [ ] Make sure that `git status` no longer shows db.sqlite3
- [ ] Push the *.gitignore* file to your work branch on GitHub

B. Create the Django Model

- [ ] Create an app called *game* and make Django aware of it, just like you were shown in the lecture

- ☐ Modify *models.py* to create a Board model (with a label, a row, a column, and a value) and a Player model (with a name, a row, a column, and a score)
- ☐ Modify *admin.py* and *settings.py* so that you can access the models from the admin app; don't forget to perform the migration steps
- ☐ Go to http://127.0.0.1:8000/admin to confirm that you can see the models

C. Create the Game Logic

- ☐ Modify the game files so that:
  - ☐ http://127.0.0.1:8000/game/create creates a 10x10 Board, with 5 Treasures and 2 Players randomly placed
  - ☐ http://127.0.0.1:8000/game displays the Board as a 10x10 grid and presents a choice to become either Player 1 or 2
  - ☐ http://127.0.0.1:8000/game/display/1/ displays the Board as well, but with up, left, right, and down buttons added that move Player 1; similarly, http://127.0.0.1:8000/game/display/2/ allows Player 2 to be moved

  Be sure to handle race conditions! Note that you can make a function atomic by placing *@transaction.atomic* above the function definition (you must also import *transaction* from *django.db*)
- ☐ The Board should display both Player scores
- ☐ Demonstrate the working game to your instructor using 2 separate browser windows, for players 1 and 2
- ☐ Push the changes to GitHub

Web App Works /10
(Instructor Stamp)

Completion

- ☐ Log out of your VM and your current machine
- ☐ Congratulations! You have completed this lab. **Be sure to hand in the zip file** and see you next week!

Important Take-Aways

- ☐ Be able to write networking code in Python using Django

Lab 8: Test a Django Server

Please see the course outline for the due date

A. Test the App

- ☐ In *226_Lab5_2022/website/game/tests.py*, create **Django** tests to test your Lab 5 code comprehensively, including ensuring that:
  - ☐ 2 Players are created
  - ☐ 100 Board tiles are created
  - ☐ 5 Treasures are created
  - ☐ movement beyond the top, bottom, left, and right borders is not possible
  - ☐ moving onto a Treasure causes the Player's score to be updated correctly
- ☐ Add, commit, and push everything to your work branch on GitHub
- ☐ Confirm that no file was missed, using:
  `git status`
- ☐ Demonstrate to your instructor that the tests are working

Local Tests Work /0
(Instructor Stamp)

B. Integrate the Tests in GitHub

- ☐ Log into GitHub
- ☐ Go to the *226_game* repository
- ☐ Click on *Actions*
- ☐ Click on *New workflow*
- ☐ Click on *Configure* under *Django*
- ☐ Change the Python versions list to contain only 3.11
- ☐ Replace

`pip install -r requirements.txt`
with
`pip3 install pipenv`
`pipenv install django`

- [ ] Find the line
`python manage.py test`
and change it to
`pipenv run website/manage.py test game`
- [ ] Click on *Commit changes...*
- [ ] In the dialog box that appears, click on *Commit changes*
- [ ] From the PyCharm Terminal, switch to the main branch and pull in the *yml* file you just created on GitHub
- [ ] Merge in your work branch and push the result to GitHub
- [ ] Confirm that the tests passed on GitHub
- [ ] Show the detailed test result on GitHub to your instructor

GitHub Tests Work /0
(Instructor Stamp)

C. Submit the Source Code

- [ ] Clean up your code to comply with Appendix A
- [ ] Push everything to your work and main branches on GitHub
- [ ] Confirm that all tests still pass
- [ ] Compress the entire *226_game* directory as a zip file (**all other formats will result in a 0**)
- [ ] Make sure the zip file is working by decompressing a copy (**broken zip files will result in a 0**)
- [ ] Submit the zip file to D2L

Completion

- [ ] Log out of your VM and your current machine
- [ ] Congratulations! You have completed this lab. **Be sure to hand in the zip file** and see you next week!

Source Code
/10

Important Take-Aways

- [ ] Be able to write tests for Django
- [ ] Have GitHub run Django tests automatically on commit

Lab 9: Deploy a Django Server

Please see the course outline for the due date

Note: This lab is based on instructions found at https://render.com/docs/deploy-django

A. Create a Render Account

- ☐ In your browser, go to https://render.com/
- ☐ Sign up for a free account using your generic ICS credentials. Do not provide a credit card

B. Change the DEBUG Flag and ALLOWED_HOSTS List

- ☐ From the main branch, create a new branch called *render*
- ☐ To make unauthorized reverse engineering more challenging, change the *DEBUG* flag in *settings.py* to
  `DEBUG = 'RENDER' not in environ`
  When deployed to Render, this will be False, causing debug mode to be turned off
- ☐ Also add

```
from os import environ, path
```
at the top

- [ ] When debug mode is off, Django requires *ALLOWED_HOSTS* to be non-empty, so add the following **underneath** the *ALLOWED_HOSTS* line:
```
RENDER_EXTERNAL_HOSTNAME = environ.get('RENDER_EXTERNAL_HOSTNAME')
if RENDER_EXTERNAL_HOSTNAME:
    ALLOWED_HOSTS.append(RENDER_EXTERNAL_HOSTNAME)
```

C. Change the SECRET_KEY

- [ ] A common security leak involves posting secret keys to Github. We actually did that when we uploaded *settings.py*! We really should read the secret key from an environment variable. For now, we will only do that if deployed on Render, so, at the bottom of the file, add:
```
if not DEBUG:
    SECRET_KEY = environ['SECRET_KEY']
```
- [ ] Now enter the following Terminal command to generate a secret key:
```
export SECRET_KEY=`head /dev/urandom | tr -dc 'A-Za-z0-9~@#$%^&*()_+={[}]|:;<,>.?/-' | head -c 50`
```
- [ ] Confirm that a key was generated by entering the command
```
echo $SECRET_KEY
```
- [ ] Ensure that your Django app still runs. You have to launch it in the same Terminal in which you created the key, or it won't work
- [ ] There are other security changes that should be made. The Django project has a deployment checklist. Some of those settings can be verified using
```
python3 website/manage.py check --deploy
```
Note that this is just a cursory check; for deployment security, a full security audit of the app and its settings is required. Having said that, avoid following these recommendations for the duration of this lab

D. Install PostgreSQL

- [ ] Instead of SQLite, we will use PostgreSQL, so enter the following commands:
```
sudo apt-get install libpq-dev
pipenv install dj-database-url psycopg2-binary
```
- [ ] At the top of *settings.py* add:
```
from dj_database_url import config
```
and at the bottom add:
```
if not DEBUG:
    db = config(conn_max_age=600, default='postgresql://postgres:postgres@localhost:5432/website')
    DATABASES['default'] = db
```

E. Install gunicorn

- [ ] For efficiency reasons, we will use the gunicorn web server instead of the built-in Django web server. Install gunicorn using the command
```
pipenv install gunicorn
```

F. Add Support for Static Files

- [ ] To provide support for serving CSS files etc., add the whitenoise package using:
```
pipenv install whitenoise
```
- [ ] In *settings.py*, look for thje MIDDLEWARE list and add:
```
'whitenoise.middleware.WhiteNoiseMiddleware',
```
- [ ] Change STATIC_URL to the following (look closely; there is a leading forward slash):
```
STATIC_URL = '/static/'
```
- [ ] At the end, add:
```
if not DEBUG:
    STATIC_ROOT = path.join(BASE_DIR, 'staticfiles')
    STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

G. Add a Build Script

- [ ] In the same directory in which Pipfile is located, create a file called *build.sh* with the following contents:
```
#!/usr/bin/env bash

pipenv run pipenv install
python website/manage.py collectstatic --no-input
python website/manage.py migrate
```
- [ ] Next, make it executable using:
```
chmod a+x build.sh
```

H. Deploy to GitHub and Render

- [ ] Add, commit, and push everything to your *render* branch on GitHub
- [ ] Confirm that no file was missed, using:
```
git status
```
- [ ] Push everything to GitHub using
```
git push --set-upstream origin render
```
- [ ] Now merge everything with the main branch and push the result to GitHub
- [ ] Make sure the tests pass
- [ ] The paid tier allows us to automate deployment via *Blueprint Instances*; however, we must stick to the free tier, so:

- ☐ create a PostgreSQL database (*New > PostgreSQL* from the *Dashboard*); the name, database, and user should all be called *website* ; no Datadog API key is required; do **not** allow connections from outside Render (you can set that once the database has been created)
- ☐ create a new web service (*New > PostgreSQL* from the *Dashboard*); connect the web service to your GitHub repository; call the site *ics___website*, where the blanks represent the 3 unique digits from your generic ICS account; the Runtime must be *Python 3*; the build command is *./build.sh* (not the leading dot!); the *Start Command* is *gunicorn website.wsgi:application --chdir website* ; under *Advanced*, add the following *Environment Variables*: *DATABASE_URL* (take the value from the PostgreSQL database that you just created; make sure there is only an internal URL, not an external one!), *SECRET_KEY* (click on *Generate* to generate a key), *WEB_CONCURRENCY* (pick 4), and PYTHON_VERSION (pick 3.11.1)

☐ Show the working game on Render to your instructor. The URL is shown on the web page that shows you the web service settings

Game on Render /10
(Instructor Stamp)

Completion

☐ Log out of your VM and your current machine

☐ **Note: You will use the Pi in the next lab. Don't forget to bring your Raspbian MicroSD card next time!**

☐ Congratulations! You have completed this lab. See you next week!

Important Take-Aways

☐ Be able to deploy a Django app via a CI/CD pipeline to Render

Lab 10: Explore IPv6

Please see the course outline for the due date

Preparation

- ☐ You will need a Pi for this Lab

A. Check Basic IPv6 Connectivity:

- ☐ Connect your Pi using the **white** Ethernet cable
- ☐ Power on your Pi
- ☐ In a Terminal on your Pi, enter the command
  `ip  add`
  and record the network interface name (probably *eth0*) and your IPv6 addresses below:
  Network Interface Name: _____
  Global IPv6 Address: 2620:0078:c000:2259:_____:_____:_____:_____
  Link Local IPv6 Address: fe80::_____:_____:_____:_____
- ☐ Now try to ping your own Pi using
  `ping6  ____`
  where the blank represents your global IPv6 address. If that doesn't work, make sure the DNS entry in /etc/resolv.conf is *nameserver 2001:4860:4860::8888*
  Ctrl+C after the first 3 successful pings
- ☐ Repeat this for the link local IPv6 address. Link local addresses (generally) need to know which interface to use, so use the command
  `ping6  ____%____`
  where the first blank represents your link local IPv6 address and the second blank represents the associated network interface name. Ctrl+C after the first 3 successful pings

- [ ] Contact another student to get that student Pi's IPv6 address and record it here:
  2620:0078:c000:2259:_____:_____:_____:_____
- [ ] Make sure you can ping the other Pi via this address
- [ ] Install the *ndisc6* package. If the installation fails, you may have to connect using a yellow cable temporarily; if so, don't forget to plug in the white cable after the installation finishes
- [ ] Attempt to contact the IPv6 router using the command
  `rdisc6 ____`
  where the blank represents the network interface name you used earlier
- [ ] Confirm that the prefix matches the first 8 bytes (recall: 1 byte = 2 hex digits) of your global IPv6 address
- [ ] Retrieve one of Google's IPv6 addresses via the command
  `host www.google.com`
  and enter the address here: _____:_____:_____:_____:_____:_____:_____:_____
- [ ] Make sure you can ping this address using *ping6*. Ctrl+C after the first 3 successful pings
- [ ] Now open a browser and go to Google
- [ ] Enter the query, "What is my IP address" and confirm that it matches the IPv6 address you recorded earlier. Note that your Pi has a globally-routable IPv6 address (unlike the 10.51.0.0/16 IPv4 address which is a local address only visible to computers in TEC 259)

IPv6 Information Correct /10
(Instructor Stamp)

Completion

- [ ] Log out of your current machine
- [ ] Shut down the Pi. If you are using a Camosun Pi, remember to remove your flash card!
- [ ] Congratulations! You have completed all labs!

Important Take-Aways

- [ ] Explore IPv6 networking

Appendix A: Code Marking Scheme

All submitted code will be evaluated based on the rubric below. Evaluation of each category starts from the bottom up and stops at the first matching level. Correctness is weighted more heavily than the other categories. When submitting compressed files, be sure to use the zip format and double-check that the compressed file can be opened, to avoid a 0 grade for the submission

Rating	Correctness/Efficiency	Documentation	Structure/Complexity
***** **perfect**	- passes all tests	- well-documented, allowing another programmer to use all functions based on the header comments alone	- well-engineered, consisting of a modular collection of simple, single-purpose functions
	- code review reveals no faults	- responsibilities of all functions are described well, without giving implementation details	- constants are used whenever appropriate
	- efficient (given the requirements)	- all parameters, return values, and side effects are explained	- globals are not used, unless unavoidable
	- no redundant operations	- comments within all functions are helpful, without being distracting, making it easy to follow along	- all constants and variables are named appropriately
			- no layout abnormalities (eg, missing or improper indentation)
			- easily used and reused
			- no undesirable side effects, such as debug output
**** **good**	- passes nearly all tests	- occasionally, there are comments that are not complete, helpful, and/or true	- largely well-engineered, except for a few, minor issues
	- a code review reveals nearly no faults; faults that are found, are minor	- could be improved by slightly reworded, slightly more, or slightly fewer comments	- a few, minor issues with constants, variables, or layout
	- generally efficient, except in a few minor cases		- easily used and reused, except in a few, minor instances
	- at most a few redundancies		- generally no undesirable side effects
*** **ok**	- passes half the tests, or more, but the failure rate is too high for a 4-star rating	- in a number of cases, comments are cryptic, false, incomplete, misleading, missing, or redundant	- in need of reengineering due to a number of issues, none, or almost none of which, are major
	- a code review reveals a number of faults, but none, or almost none of them, are major		- on a number of occasions, there are issues with constants, variables, or layout
	- somewhat efficient, but there are a small number of major inefficiencies		- often easily used and reused, but there are a small number of major problems, none of which render the work unusable

Rating	Correctness/Efficiency	Documentation	Structure/Complexity
	- potentially many redundancies		
** **fail**	- fails more than half the tests	- only little relevant documentation	- a number of major issues, requiring major changes
	- a code review reveals a high number of faults, including a number of major faults	- comments are often cryptic, false, incomplete, misleading, missing, or redundant	- not easily used and reused
	- not efficient, although slow progress is being made		
* **fail**	- fails nearly all the tests	- essentially no legitimate documentation	- only a few functions, many of which are responsible for too many things
	- code review reveals a proliferation of major faults		- essentially not usable or reusable
0 **fail**	- fails all tests and/or does not run; source cannot be viewed	- no legitimate documentation and/or does not run; source cannot be viewed	- no legitimate code and/or does not run; source cannot be viewed

Correctness/Efficiency: _____/ 5 x 2 = _____/ 10
Documentation: _____/ 5
Structure/Complexity: _____/ 5
Total: _____/ 20 (this ratio will be used to determine the applicable point score)