Lab 12: Project

CSE/IT 107

NMT Computer Science

> "I wanna be the very best,
> Like no one ever was.
> To catch them is my real test,
> To train them is my cause."
>
> — Pokémon Intro Song

> "So, tell me about yourself. Are you a boy or a girl?"
>
> — Professor Oak

---

1 Introduction

This will be your final lab for the semester.

How very spooky.

2 tkinter

tkinter is a basic graphics library for Python. You will be using it as your user interface for this lab. Here we will be going over a few examples of you can use tkinter to make some simple user interfaces. There are tons of options in tkinter that we will not have space to cover, so if there is something you want to do that is not covered here, google around and check if someone's done what you want before – just make sure to cite them in your comments!

To start with, we need to create a Tk object. Tk is a class contained in the tkinter module of Python that allows us to easily open new windows. The simplest way to create a tkinter window is shown in the code below, with the result in Figure 1.

```
>>> import tkinter
>>> window = tkinter.Tk()
```

Listing 1: Just a tkinter window. See Figure 1.

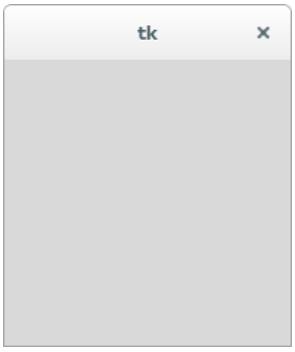

Figure 1: Basic tkinter window. See Listing 1 for the code.

An empty window isn't particularly impressive, so let's add some buttons to the window. In order to add a button, we need to instantiate a new object of tkinter's Button class.

```
>>> import tkinter
>>> window = tkinter.Tk()
>>> button = tkinter.Button(window, text='Click me!', command=lambda: print("Hi!"))
>>> button.grid(row=0, column=0)
>>> window.geometry("400x400")
''
```

The button's constructor should be fairly readable: the first argument is the window or frame that the button will go inside, the text option specifies what the button says, and the command option is the function that will be called when the button is clicked. In this example we've created a simple lambda function, but in most cases you will want something a bit more complicated than a lambda function. In that case you'll just want to pass an already defined function.

The next line determines how the button should be laid out inside the window. tkinter provides for multiple layout methods, but for this example we will be using `grid`. `grid` allows us to easily specify where widgets go relative to one another while letting tkinter handle their exact sizing for us. A widget can be a bunch of different tkinter objects: text, a button, a label, a frame, ...

Finally, the `window.geometry("400x400")` statement tells tkinter to make our window 400 pixels by 400 pixels. This is not strictly required as otherwise tkinter will autosize our window based on how big its elements are.

If we want to do a slightly more complex layout, we can specify multiple buttons in different rows and columns:

```
>>> import tkinter
>>> window = tkinter.Tk()
>>> tl = tkinter.Button(window, text="Top left")
>>> tl.grid(row=0, column=0)
>>> tr = tkinter.Button(window, text="Top right")
>>> tr.grid(row=0, column=1)
>>> bl = tkinter.Button(window, text="Bottom left")
>>> bl.grid(row=1, column=0)
>>> br = tkinter.Button(window, text="Bottom right")
>>> br.grid(row=1, column=1)
```

Listing 2: Multiple buttons in tkinter. See Figure 2

Figure 2: Basic tkinter window with grid layout. See Listing 2 for the code.

For a more complicated example, we will show a short program that uses Frames to swap between different sets of buttons. A frame is a widget that can be used to group other widgets together. We can add widgets to a frame, then add that frame to the window in order to display all of the widgets it contains.

```
import tkinter

#remove one frame, add another
def add_remove(to_remove, to_add):
  to_remove.grid_remove()
  to_add.grid(row=0, column=0)


def main():
  window = tkinter.Tk()

  #Create frames.
  left_frame = tkinter.Frame(window)
  mid_frame = tkinter.Frame(window)
  right_frame = tkinter.Frame(window)

  #Create button for left-most frame.
  tmp_button = tkinter.Button(left_frame, text='Go right.',
    command=lambda: add_remove(left_frame, mid_frame))
  tmp_button.grid(row=0, column=0)

  #Create buttons for middle frame.
  tmp_button = tkinter.Button(mid_frame, text='Go left.',
    command=lambda: add_remove(mid_frame, left_frame))
  tmp_button.grid(row=0, column=0)
```

```
26    tmp_button = tkinter.Button(mid_frame, text='Go right.',
27      command=lambda: add_remove(mid_frame, right_frame))
28    tmp_button.grid(row=0, column=1)
29
30    #Create button for right-most frame.
31    tmp_button = tkinter.Button(right_frame, text='Go left.',
32      command=lambda: add_remove(right_frame, mid_frame))
33    tmp_button.grid(row=0, column=0)
34
35    left_frame.grid(row=0, column=0)
36
37    window.mainloop()
38
39  if __name__ == '__main__':
40    main()
```

Try running this code to see how it behaves. Everything here has been previously introduced, except for the `.grid_remove()` and `.mainloop()` methods. `.grid_remove()` removes a widget from its parent without necessarily getting rid of the widget.

`.mainloop()` is a bit more complicated. Basically, it waits for events to happen to your window and handles them as they occur. It does not end until the window is exited. If you don't include `.mainloop()` somewhere in your program, it will simply exit after a moment of the window popping up. Thus, a programming using tkinter will generally set up the window and anything else it needs, then call `.mainloop()` and handle anything else in the commands tied to buttons.

Buttons and frames are not the only types of widgets that tkinter supports. There are other kinds of widgets such as text or labels. Here is an example:

```
1  import tkinter
2
3  window = tkinter.Tk()
4  window.title("Title!") # Notice dat title!
5  window.config(background = "white")
6
7  left_frame = tkinter.Frame(window, width = 200, height = 400)
8  left_frame.grid(row = 0, column = 0, padx = 10, pady = 3)
9
10 right_frame = tkinter.Frame(window, width = 200, height = 400)
11 right_frame.grid(row = 0, column = 1, padx = 10, pady = 3)
12
13 helloworld = tkinter.Text(left_frame, width = 30, height = 10)
14 helloworld.insert(tkinter.END, "Hello World on the left")
15 helloworld.grid(row = 0, column = 0)
16
17 helloworldr = tkinter.Text(right_frame, width = 30, height = 10)
18 helloworldr.insert(tkinter.END, "Hello World on the right")
19 helloworldr.grid(row = 0, column = 0)
20
21 window.mainloop()
```

Listing 3: tkinter Hello World in two columns. See Figure 3.

Figure 3: tkinter with text. See Listing 3 for the code.

This is the basic flow to how using tkinter works. There are lots of other widget types you may find useful, check out the official documentation at

`https://docs.python.org/3/library/tkinter.html`,

or the NMT-based semi-official documentation written by the wonderful John Shipman at

`http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html`.

We recommend the official documentation for three reasons: it is authoritative, it is good, and it teaches you to learn how to read the official Python documentation. However, John Shipman's tkinter reference is also excellent and we would be very wrong not to mention it here.

You will have to use these documentations to figure out how to display certain text or images to the tkinter window. This section was only intended as a small introduction to the principles of tkinter – windows, frames, buttons, widgets and their placement – while you will have to do just a little bit of research on how to do more.

Python 2 vs Python 3

tkinter changed a few small things between Python 2 and Python 3. Note that when you see it called *Tkinter* with a capital T, you are looking at documentation about the Python 2 version, while *tkinter* is the Python 3 version.

Most of the documentation that you will be looking up will be the same for Python 2 and Python 3 tkinter though. Tkinter only changed some names between the Python versions, while the usage is still the same.

3 Project

Boilerplate

Remember that this lab *must* use the boilerplate syntax introduced in Lab 5.

For this assignment, you will be creating a multiplayer game similar to the combat system of the Pokémon or Final Fantasy games. You will be writing two programs, though you are free to have them share some amount of code. Your two main files should be named battle.py and battle_server.py, though you are encouraged to have more than just those two files.

Some files have been provided for you on Canvas. We recommend using them as a starting point, though you are free to completely ignore them.

Figure 4: Pokémon battle screen. (Source: Wikipedia)

3.1 Game

The game you are creating will be a two-player battle game. Each player will have a "party" of monsters. At the start of the match, each player will choose one of their three monsters to become active. Then a series of turns will occur until one player's entire party of monsters has been defeated.

3.1.1 Turns

A turn consists of each player choosing one action to perform. The possible actions fall into two categories: switching the player's active monster or using a move from the currently active monster.

If a player chooses to switch their active monster, they must then choose from their current inactive and undefeated monsters. The selected monster will become active while their current active monster will become inactive.

If a player chooses to use a move from their currently active monster, they will then choose one of up to four moves that their monster knows.

Once both players have chosen a move for the turn, the moves will be executed. If both players chose to switch monsters, then the order that the switches occur does not matter. If one player

chose to switch their monster while the other chose to use a move, then the switching player's action will go first. If both players choose to use a move, then the player whose monster has a higher speed stat (see Section 3.1.2) will perform its move first. The second monster will only perform its move if it is still undefeated after the first monster performs its move.

If at the end of a turn either or both monsters have had their health reduced to 0 or less, then that monster has been defeated. Its user must immediately switch out their monster for one that has not been defeated. This switch occurs before the start of the next turn. If no undefeated monsters remain, that player has been defeated. If both players are defeated at once, the match ends in a draw.

3.1.2 Monsters

Each monster will have a number of stats:

Attack Indicates attack power of the monster. The base damage of any attack Move that this monster uses is modified by

$$base\_damage \cdot \left(1 + \frac{attack}{100}\right)$$

Defense Indicates defensive strength of the monster. Any incoming damage to the monster is reduced by

$$damage \cdot \left(1 - \frac{defense}{100}\right)$$

A monster is not allowed to have a Defense greater than 75.

Health The maximum health that a monster has. Any time it is attacked the monster's current health will be reduced, thought the maximum will stay the same.

Speed Indicates the speed of the monster. Is used to determine which of the active monsters will use its Move first.

Name Each monster must have a name so that you can properly empathize with its plight.

In addition to these stats, each monster can have up to 4 moves, as defined in Section 3.1.3

3.1.3 Moves

A move is an action that a monster can perform when it is active. There are four types of move:

Offensive An offensive move deals damage to the opponent's active monster. Each offensive move has a base damage stat that helps determine how much damage it does, based on the two formulas in Section 3.1.2.

Heal A healing move restores health to the monster making the move. A healing move has a stat indicating how much health it will restore. A monster cannot be healed above its maximum health.

Buff A move that temporarily increases one of the active monster's stats. A buff move has three attributes: the stat it increases (attack, defense, or speed), the amount it increases that stat by, and how many turns it lasts. A buff can allow a stat to be increased beyond the point it normally could be.

Debuff A move that temporarily decreases one of the stats of the opponent's active monster. Has the same attributes as buff, but decreases instead of increases.

In addition to the above mentioned attributes, each move must have a name.

3.2 Client (battle.py)

You will need to write a client using tkinter that provides a nice user interface to those playing the game. You should model your interface off of Figure 4, though you are free to modify it as you see fit. The interface needs to do the following:

- Display the name, current health, max health, attack, defense, and speed of the currently active monsters from both players.
- Display information about what is happening in the game; i.e. what moves each monster is using, what the results of those moves are, what monsters are switched for.
- Allow the user to choose between their various options each turn, as shown in Section 3.1.1.

In addition, your client will need to connect to your server in order to send the actions of its player and receive the results of each turn. It will also need to receive information about the opponent's monsters at the start of a game.

When the client is first started, it must load the information about its party of monsters from a text file. This information must then be sent to the server upon connection.

3.3 Server (battle_server.py)

You will need to write a server that allows two clients to connect to it in order to play the game. This server will carry out the majority of the game's logic and handle sending and receiving information from the players. It will need to fulfill the following tasks:

- Receive information about each player's party, store that information, and send it to each of the players as necessary.
- Wait for each player to select an action, then calculate the outcome of both actions.
- Detect when one player has won and inform each player.
- Anything else you find would be handy for the server to take care of.

You are free to use whatever protocol you wish for communicating data between the server and the clients.

3.4 Terms

Active The current "fighting" monster. Each player can only have one active monster at a time.

Damage The amount subtracted from a monster's current health after it is attacked. The full formula for damage is

$$base\_damage \cdot \left(1 + \frac{attack}{100}\right) \cdot \left(1 - \frac{defense}{100}\right)$$

Defeated A monster that has had its health points reduced to 0. A defeated monster cannot be active.

Monster A mysterious creature that we are using to fight for morally questionable reasons (Section 3.1.2).

3.5 Extra Credit

Any or all of these can be included in your lab. If you do extra credit, include a file called README.txt that details which extra credit opportunities you did. If you did something extra not mentioned here, also mention it in your README.txt. If you have any ideas for extra credit that aren't mentioned here, email rwhite00@nmt.edu.

3.5.1 Awesome Name (5 points)

Give your game a REALLY AWESOME NAME!

3.5.2 Status Effects (10 points)

Implement status effects into some of your moves. That is: add some moves that have a chance to inflict effects beyond damage and debuffs onto the enemy. For example:

Poison A poisoned monster takes some amount of damage each turn for a set number of turns.

Paralysis A paralyzed monster has a chance to not perform a move when it is selected. Additionally, a paralyzed monster will always move second if the other monster is not paralyzed.

Scared A scared monster has a chance to ignore a move command and instead switch itself with a random inactive undefeated monster.

3.5.3 Types and Strengths/Weaknesses (15 points)

Implement elemental types for your monsters. This means that each monster will have some kind of alignment. Examples might be fire, water, flying, or jello. Define (preferably in a file) weaknesses for the different types. For example, water types might deal 150% normal damage to fire types, while only dealing 25% normal damage to jello types. See Figure 5.

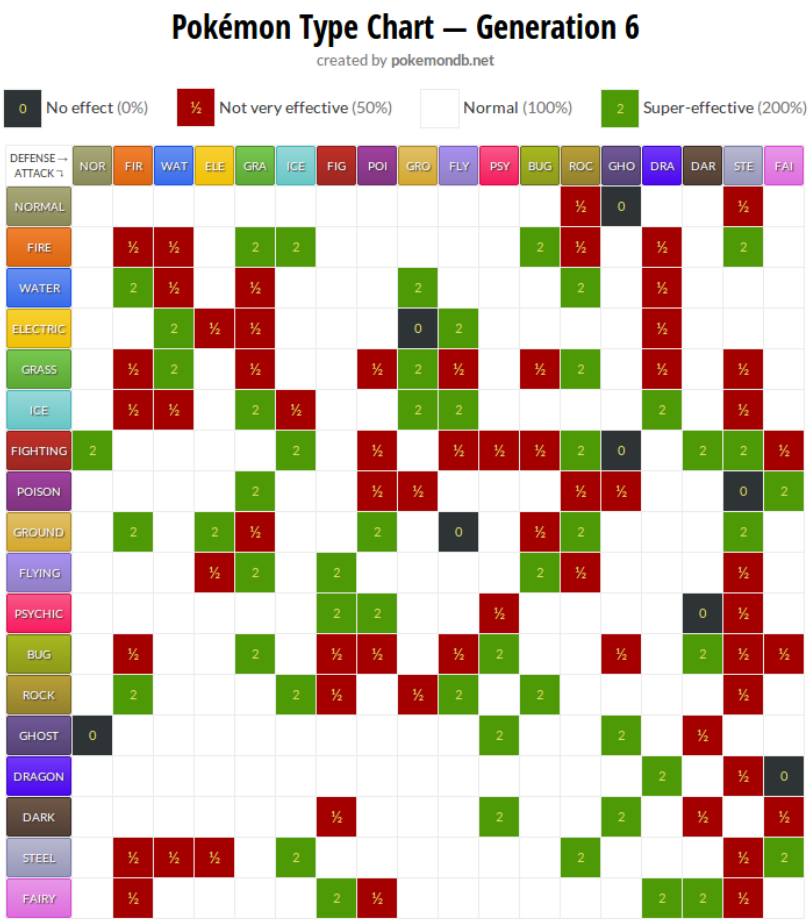

Figure 5: Type strengths and weaknesses in Pokémon.

3.5.4 Validate Monster Stats (10 points)

Have your server validate the monsters sent in by the client to make sure they aren't too overpowered. Make up your own rules for what determines if a monster is valid. An example might be that the sum of all monster attributes must be less than 250. If an invalid monster is sent to the server, send a rejection message and either disconnect the client or request a new valid monster.

3.5.5 Larger Party Size (10 points)

Allow each player to have up to 6 monsters in their party instead of just 3.

3.5.6 Monster Select (10 points)

When the client starts and loads up a group of monsters, allow the users to select which of the monsters they wish to use in the GUI.

3.5.7 Monster Graphics (15 points)

Let a monster have an associated image that is displayed on both clients. To do this, you will need to have the client load an image, then send that image to the server so that the server can pass it on to the other client (remember that in a real online game your clients won't have access to the same files to read from, so you can't just have them both load from the same file).

If you use images from online, make sure to cite where you got them!

3.5.8 Randomly Generated Monsters (10 points)

In addition to loading monsters from files, add the ability to randomly generate a monster. You should make your randomly generated monsters have some logic to how their stats are created so that they are not too over- or underpowered. For examples, see Section 3.5.4.

3.5.9 User-Created Monsters (15 points)

In addition to loading monsters from files, provide an interface to allow a player to create their own monsters. Let them choose the name, stats, and whatever else you can think of for their party of monsters.

3.5.10 Sound (10 points)

Let sounds play in response to certain events. Have sound effects for different moves, monsters being defeated, switching out monsters, and whatever other events you want to tie sounds to.

If you use sounds from online, make sure to cite where you got them!

3.5.11 Critical Hits (10 points)

Allow an offensive move to have a chance to make a critical hit. For each offensive move, give it a number from 0 to 100 indicating how likely it is to critically hit. Each time the move is performed, randomly determine – based on that percentage – to get a critical hit. If a move gets a critical hit, it should do twice as much damage as normal.

Optionally, give each monster a built-in crit chance. That is, if a monster has a built-in crit chance of 5%, then all of its offensive moves have an extra 5% chance to get a critical hit, even if their previous chance was 0%.

3.5.12 Other Extras (?? points)

If you have any ideas for more extra credit opportunities, email Russell and he will probably add them to the lab document.

4 Submitting

Files to submit:

- battle.py (Section 3)
- battle_server.py (Section 3)
- Any other files created. (Section 3)

You may submit your code as either a tarball (instructions below) or as a .zip file. Either one should contain all files used in the exercises for this lab. The submitted file should be named either `cse107_firstname_lastname_lab12.zip` or `cse107_firstname_lastname_lab12.tar.gz` depending on which method you used.

For Windows, use a tool you like to create a `.zip` file. The TCC computers should have `7z` installed. For Linux, look at lab 1 for instructions on how to create a tarball or use the "Archive Manager" graphical tool.

Upload your tarball or .zip file to Canvas.