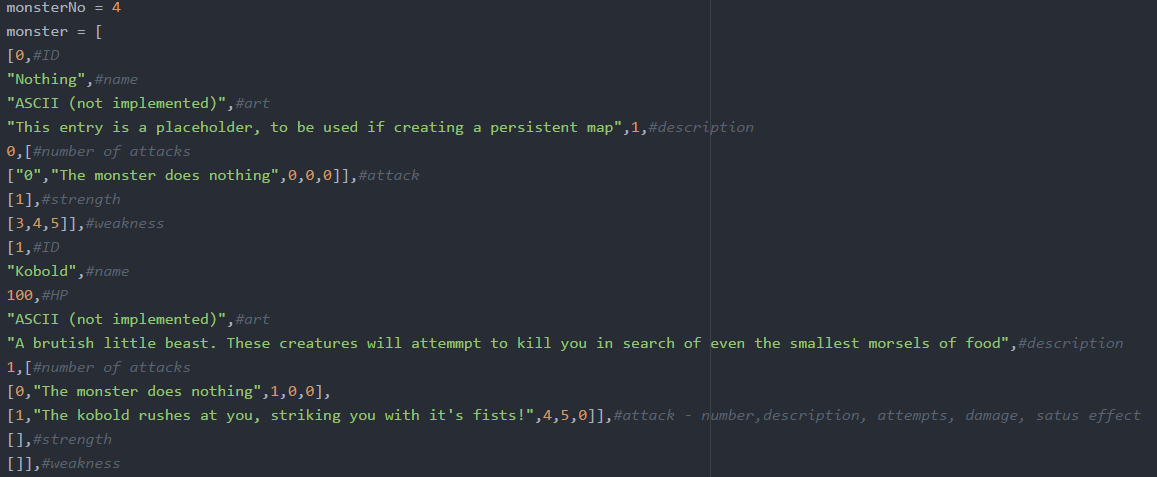
**Chapter 6 – Fight Part 1**

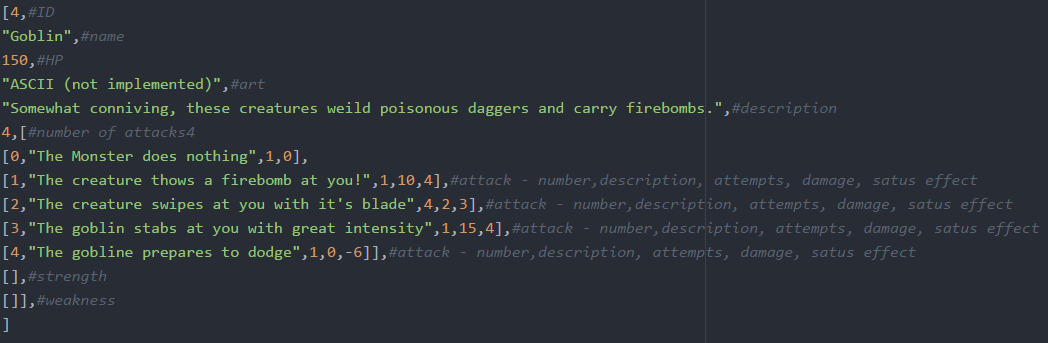
As stated in the previous chapter, this chapter only requires a firm understanding of everything taught so far. The reason being is because we want you to focus more on how the logic works rather than feeling frustrated because you don’t understand what a particular line means. This allows you to focus more on why the program works and makes the process more enjoyable.

Before we begin coding, it is a good idea to start thinking about two main things – what type of attacks you classes will have and what type of monsters your game will have.

The monsters will have the following main attributes in their lists:

* an ID number so that we can refer to them easily in a game
* the name of the monster which the player will see
* the HP of the monster
* ASCII art (this is an extension)
* A monster description
* The number of attacks
* A list of each attack
* A list of strengths
* A list of weaknesses

Here some examples showing what we mean by the above attributes:

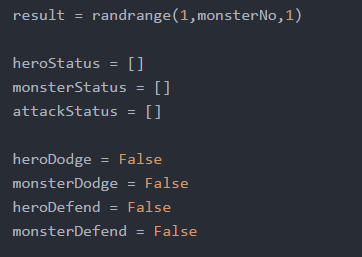


Each attack also has extra details which have been outlined in the comments as attack ID, description, the number of strikes, damage per strike and any status effects from the attack. It is important the status effects have the same effects as our items, hence why we used 3,4 and -6.

(the reason we use a negative here is to show it’s an effect the monster uses on itself)

Good planning is always required when writing a program and doing this before you start will minimise any problems along the way. This way it is easier to tell if a problem is coming from the way you designed something or the way you implemented something.

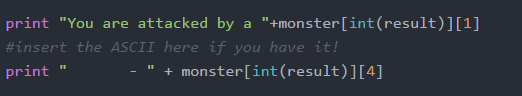
First thing first is to declare the function and the required global variables in the function. We then initialise a few variables like this:

The first line simply uses the random number generator to choose a monster from our list.

We also create blank lists for hero statuses, monster statuses and attack statuses. These will become more clear as progress.

We also have some Booleans for dodge and defend statuses of both the player and monster.

As soon as the fight starts it would be a good idea to tell the player the name of the onster they are facing and give them a short description. This is fairly simple using what we learnt from **chapter 5**:



(If you have drawn the ASCII for the monster, you would also put it here)

We will also retrieve the monster HP using the following line:

Screen Clipping

For the whole fighting process we will use a while loop but since there be many conditions to check for, our while loop will be manage differently. We will be using returns within the loop to signal when we want to exit the loop but we must remember to be careful as if we don’t cover every possibility, our fight could go on forever. Therefore we will just start our while loop like this:

While True:

For the fight function, we will split it into three parts: the processing of damage taken, the processing of status effects and the processing of user inputs. In this chapter (part1) we will only look at processing of damage taken. Part 2 will cover the last two aspects.

Processing of damage taken

We will break this into three sections where we consider if the player has a dodge effect active, if the player decided to block and if the player decided to attack normally.

The first thing we do is select one of the monsters attack at random and we must remember this option. This is done with the following line:

Screen Clipping

(remember to add 1 to the upper bound because we want the chance of the monster also doing nothing)

Once we have chosen our attack we need to create a for loop for which goes through each strike of the attack. Now be very careful about how the loop is set up as this information will be stored in nested lists if you refer to the top of this chapter. Don’t worry though, we will break it down so you don’t get lost:

Screen Clipping

It would be easier to understand if we space out each index here:

monster [int(result)] [6] [int(monsterAttack)] [2]

* So first we write monster to show we are referring to the monster list.
* The first index refers to the specific monster we have chosen. If you look back you will see that we randomly generated this.
* The next index (6) points to the nested list for that specific monster which contains each attack
* The 3rd index points to the random attack we chose before executing the for loop
* The final index points to the number of strikes for that specific skill

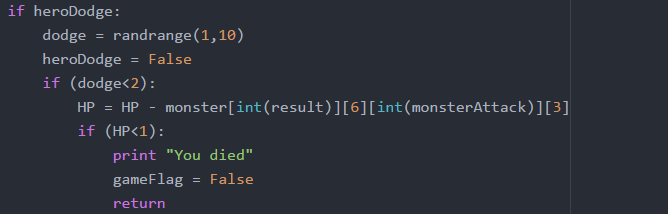
In the end, this for loop should repeat ‘x’ times where ‘x’ is the number of strikes.

Next we will add a chance that our monster will miss which is very simple and is up to you as the designer what value you want this to be:

Screen ClippingNOTE: if you don’t need an integer you can just only have 2 parameters in randrange().

So I have decided to give our monster a 20% chance of missing (4/5) and now I will write an if statement stating what happens if the monster doesn’t miss.

Now we will consider if the player has a dodge effect active. If you remember at the beginning where we had a Boolean for dodge status, this is where we will use it. If the dodge status is true, there will be a high chance the attack misses. Here we give an 80% chance to dodge:



If the attack is dodged, the player takes damage equal to the value which we defined in our monster list. The new HP of the player becomes HP – damage.

We then have an if statement to check if the player has less than 1HP. If so we print a death message and set the game flag to false. The game flag will be important later on. Here is our first use of ‘return’ as since our player has died, we can exit the fight loop.

\*IMPORTANT NOTICE\*

This is the most complicated part of the function

Since our status effects are based on items that already exist in our game, we are going to keep a record of which status effects are active in a list of lists.

Any status effects that negatively impact the player are stored in ‘heroStatus’ and anything that positively affects the monster is stored in ‘monsterStatus’. We also made this easy to sort by using positive and negative numbers in our monster list. This how it is done:

if (monster[int(result)][6][int(monsterAttack)][4]<1):

monsterStatus.append([0-monster[int(result)][6][int(monsterAttack)][4],loot[0-monster[int(result)][6][int(monsterAttack)][4]][3],loot[0-monster[int(result)][6][int(monsterAttack)][4]][4]])

elif(monster[int(result)][6][int(monsterAttack)][4]>0):

heroStatus.append([monster[int(result)][6][int(monsterAttack)][4],loot[monster[int(result)][6][int(monsterAttack)][4]][3],loot[monster[int(result)][6][int(monsterAttack)][4]][4]])

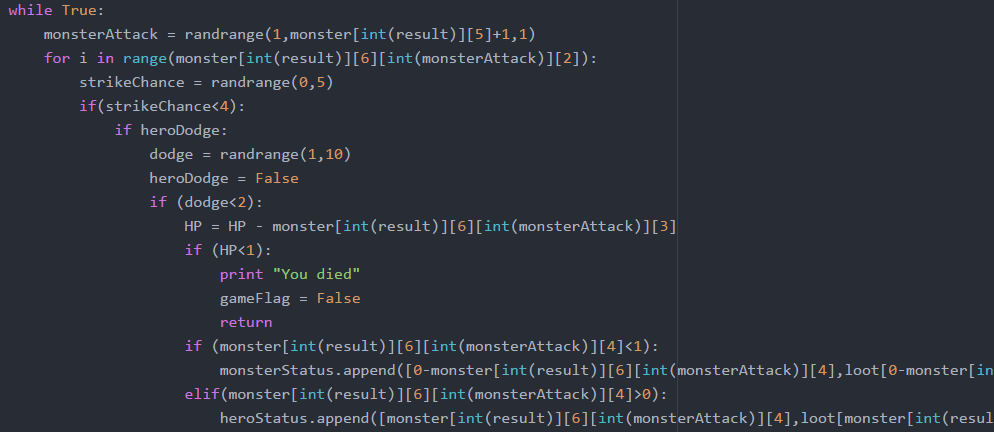
Since the above if statement is long and complex, we have colour coded it to make it easier to break down:

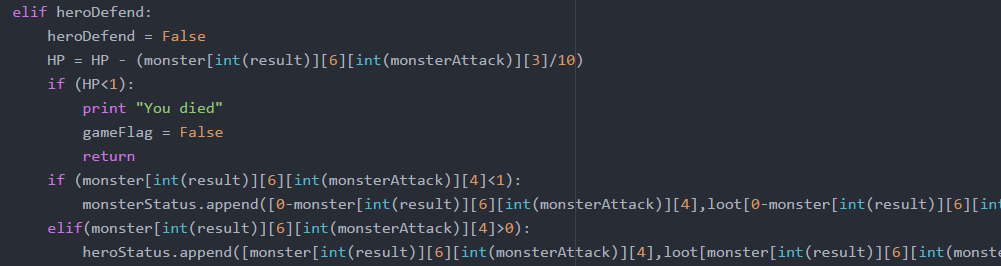
* Yellow – this if statements just checks positive or negative to we know who this benefits
* Green – To our list the first element we will include the ID of the effect. If it is negative we must remember to make it positive since our item IDs are all positive. This simply done by doing 0-number.
* Turquoise – Using the ID from the green section, we find the item in our loot list and add the effect duration.
* Teal - Using the ID from the green section, we find the item in our loot list and add the effect damage.

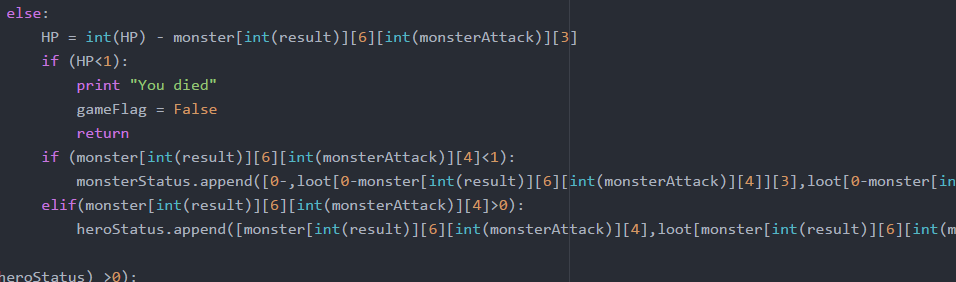
So essentially what we have done is appended a list with 3 values. (simply put we are doing XStatus.append([x,y,z]) as in the end its just 3 numbers we retrieve.)

The rest of this chapter is straight forward as we will mainly be reusing code fragments already used in this chapter.

To make sure we are on track, compare what you have so far with our example file. The indentation of if statements have to be precise. For now here is a picture:

(to save space the long lines have been cut off)

Now we will continue within the ‘strikechance’ if statement as we still have to do something if the player defends. Following the same principles as before we have written the following with the only difference being that if a player blocks, they take 10 times less damage. Therefore in our HP, we divide the damage by 10 before subtracting. Everything else that follows is the same so we can copy past scenarios for HP being less than 1 and status effects:

And our final case is when nothing happens and therefore we can just copy paste what we have written before and looks like the following:

That concludes the first phase – processing of damage taken.

There is a large amount of information to take in and it is recommended that you compare your code with our example files and make sure that you FULLY understand everything we have created in this chapter.