Canberra Girls' Programming Network

Scissors-Paper-Rock-Lizard-Spock Workbook

Now that you have the basic version of your game working, you can start to add some extensions to it. These extensions have been listed roughly in order of easiest to hardest, but you can complete them in any order you like. Some of these extensions are quite challenging. You may need to refer to the Python documentation (https://docs.python.org/3/) to complete them. Or, of course, ask a tutor for help!

1. Who's winning overall?

If you're playing many rounds with the computer, it would be fun to see how often you win! In this section we'll introduce counters to determine what number of games have been won by the human and the computer in each round, and how many rounds have been won and lost overall.

Extension 1: Running tally

1. Think about what you want to count: The number of ties; the number of times the human wins; the number of times the computer wins; or all three? Per round; overall; or both? Make enough variables for you to count what you want.
2. Initialise your counters. What value should your counters start at?
3. Determine where in your code to increase your counters. So far we haven't worked out who has won or lost each round. How could you do this? (Hint, you may need the > operator.)
4. Include a print statement at the end of each round, which prints out the result of the round.
5. Include another print statement when the human exits the game, which prints the overall number of rounds won and lost.
6. As an optional extra, for 4 and 5 above, print out the percentage of games won by the human!

2. Let's fix some bugs

In Exercise 5, you worked out a few ways that you could break your program. If you have been testing your program since then, you might have found a few more ways to break it. Now we're going to fix some of those things and make the code more robust to different types of inputs.

There are a few little issues with our code right now if the user:

- enters a move with any uppercase letters,
- enters a move that is not valid,
- does not enter a move at all,
- enters something other than "yes" or "no" when asked if they want to play again,
- enters a negative number of games per round, or zero.

We need to make our code more robust! Let's see what we can do to fix these issues!

Extension 2: Make the inputs case insensitive

So far we have talked about variables containing text, numbers or lists. The correct name for variables that contain text in Python (and most other languages) is **strings.**

When Python compares strings, the case of the letters are important. Humans understand that "Rock", "ROCK" and "rock" have the same meaning. But in Python, "Rock", "ROCK" and "rock" are all different strings. So far, you have to make sure you type the moves and the response to the question to play again in lowercase letters.

Python's strings have some functions that can help with comparing strings with different cases. One of these is `lower()`. It converts a string to all lowercase. You call it like this.

```
dinner = "cHicken"
dinner_lowercase = dinner.lower()
if dinner_lowercase == "chicken":
   print("We're having chicken for dinner.")
```

This, of course, will print "We're having chicken for dinner.". Notice the dot when calling `lower()`! That's important. We're going to fix our program so that it works with inputs containing any case.

1. Use the `lower()` function to convert the human's move to lowercase before comparing it to the computer's move.
2. Do the same for the human's answer to whether they want to play again.
3. Test your code by changing the case of the human's moves. For example if you type "Rock", "ROCK" or "rock" for the human's move, it should now work all the same way!

Extension 3: Check that the human's move is valid

We still have some problems in our code to fix. If we enter something weird like "batman" or "I like fruit" for the human's move, the program doesn't really know what to do with it. In this extension we're going to compare the human's move to a list of allowed moves and print an error message if the human types an invalid move.

If you have a list, you can test if that list contains a value by using the `in` keyword. For example:

```
dinner_options = ["lasagne", "casserole", "roast lamb"]
your_selection = input("What do you want for dinner?")
if your_selection in dinner_options:
    print("Yes, you can have that for dinner.")
else:
    print("Sorry, that is not available.")
```

You can also use `not in` to check if something is not in a list.

```
dinner_options = ["lasagne", "casserole", "roast lamb"]
your_selection = input("What do you want for dinner?")
while your_selection not in dinner_options:
    your_selection = input("That is not available. Please choose something else.")
```

Remember we already made a list of allowed moves in module 4 of the first workbook. Let's now check that the human's move is one of those allowed moves.

1. If the move made the human is not an allowed move, print a warning and ask the user to enter a new move. Think about:
    a. Should this come before or after you called `lower()` on the human's input?
2. As an optional extra, do the same thing for the human's answer to whether they want to play again. You should make "yes" and "no" the only valid answers.

Remember to test your code!

Extension 4: Check that the number of games per round is valid

We still have one last problem we need to fix in our code. If the user enters a negative or zero number of games per round, the program doesn't know what to do. And it can't, because that doesn't make sense! In this extension we're going to make sure that the number of games per round is a positive. We can check if a number is negative or zero by using the less than or equal to operator, <=. For example:

```
number = -3
if number <= 0:
    print("The number must be positive.")
```

1. Use the less than or equal to operator to check if the user has entered a non-zero, positive number of games per round.
2. If they haven't, print a warning and ask them to enter a new number.

Test it out with some different numbers!

3. Let's get organised

In this extension we'll look at making our code a bit more organised and a little easier to read. One way to do this is called refactoring. To refactor code, we can take a chunk of the code that does a specific job and put it in its own function.

So far, we've been calling functions in our code that have been written by other people. Now, we're going to learn how to write our own functions.

A function is a piece of code that is separate to the main flow of our program. An example of this is the `input()` function we used in the first workbook. A function can receive information through *arguments*. In the case of the `input()` function, the argument was the text we typed in between the parentheses. A function can also return information to us to be used or stored if required. The `input()` function returns the text the user typed in the shell.

Here is an example of creating a function:

```
def is_on_the_menu(your_selection):
"""
This is a special sort of comment called a docstring. We write what the
function does so that we, or someone else, can easily look it up later!
"""
    # don't forget to indent inside your functions!
    dinner_options = ["lasagne", "casserole", "roast lamb"]
    if your_selection in dinner_options:
        # it's on the menu! Give back True
        return True
    else:
        # it's not on the menu, give back False
        return False
```

The function works out if a dinner option is available. The function is called `is_on_the_menu`, and takes one argument called `your_selection`. Then it checks if `your_selection` is in the `dinner_options` list. If it is, then it returns True, if it's not then it returns False.

Here is an example of how we could use this function.

```
your_selection = input("What do you want for dinner?")
if is_on_the_menu(your_selection):
    print("Yes, you can have that for dinner.")
else:
    print("Sorry, that is not available.")
```

If someone wants to find out what our function does later, they can just write:

```
print is_on_the_menu.__doc__
```

And our program will print out the docstring we wrote (yours should be more helpful than this one!).

Extension 5: Refactor into some functions

Now we need to have a look through our code and see what we can refactor into its own function. Look for parts that do a job all on their own... One good example is generating a move for the computer. There we use *random* and produce a move. Maybe we could create a function there that just gives us back a move!

Things to remember:

- A function doesn't always have to have arguments.
- If we want something back we need to put it in a variable. Make sure you call the function in a variable assignment if you're expecting something to be returned. *Eg. fruit = random_fruit_generator()*
- Don't forget to indent!
- Ask one of the tutors if you get a bit stuck :)

Once you've completed your first function, look through your code and see if you can spot any more places where we can refactor. *Hint: what about deciding who wins?*

See how much easier it is to read now? Do you think that it will be easier to test code in functions? How about reusing code?

4. Scissors, Paper, Rock, Lizard, Spock!

In this extension we'll be extending the rules for scissors, paper, rock to include lizard and Spock! You'll be modifying the code you've already written to include two new elements. You've done this for three moves, so let's see how it changes with five moves.

The Rules:

- Scissors cuts Paper
- Paper covers Rock
- Rock crushes Lizard
- Lizard poisons Spock
- Spock smashes Scissors
- Scissors decapitates Lizard
- Lizard eats Paper
- Paper disproves Spock
- Spock vaporizes Rock
- Rock destroys Scissors

Wow, that's a lot of rules. And a lot of `if` statements. Maybe there's a better way, now that the rules have become so complex. To do this efficiently, we're going to need a **dictionary**.

A dictionary is a collection of things, like the lists we discussed earlier. The major difference is that each item in a dictionary has a *key* and a corresponding *value*. You get a value from a dictionary by passing it a key. It'll make more sense if we look at an example.

```
phonebook = {"mary" : 0492281238, "frank" : 0428195473}
print("Mary's phone number is", phonebook["mary"])
```

In this example, `phonebook` is the dictionary. It contains the names of some friends (`"mary"` and `"frank"`) and their phone numbers (`0492281238` and `0428195473`). The names are the keys, and the phone numbers are the values. We can look up Mary's phone number by typing `phonebook["mary"]`. Take note of of all the different syntax here, because it's important!

Remember how we said before that you can create lists of lists? Well you can also create dictionaries of lists! What would that look like?

```
menu = {
    "breakfast" : ["croissants", "toast"],
    "lunch" : ["sandwiches", "wraps"],
    "dinner" : ["pizza", "stir-fry"]
}
```

Now, the keys are different meals, and the values are lists of menu items for that meal. So how would we use something like this? We could ask someone what meal they wanted to eat and then check whether it is on the menu we created.

```
1  meal_chosen = input("Would you like breakfast, lunch or dinner? ")
2  dish_chosen = input("What would you like for " + meal_chosen + "?")
3  if dish_chosen in menu[meal_chosen]:
4      print(dish_chosen + " is on the menu, we will serve you soon.")
5  else:
6      print("I'm sorry, that dish is not on the menu.")
```

In this example we have asked the user if they're having breakfast, lunch or dinner, and then what dish they want to eat. Then, in line 3 we do several things. When we call `menu[meal_chosen]` we are asking for the list corresponding to the `meal_chosen`. For example if the user selects `"lunch"`, `menu[meal_chosen]` will return `["sandwiches", "wraps"]`. Then, we use the `in` keyword to check if the `dish_chosen` is in the list (if you didn't do Extension 3, you may want to read it now to learn about how the `in` keyword works). If it's in the list, then we know that the dish is on the menu!

Extension 6: Replace `if`s with a dictionary

1. Make a dictionary. They keys should be each of the possible moves, and the values should be a list of all the other moves that that move would beat. For example, one item in your dictionary will be
   `"scissors" : ["paper", "spock"]`
   because scissors beats paper and spock.
2. Replace your if statements with a new if block that checks
   a. If the human's and computer's moves are the same, it's a tie.

    b. If the computer's move is in the list of moves that the human's move beats, human wins.
    c. Otherwise, computer wins.
3. Update any other parts of your code you need to to add the new moves, for instance the list you added in Exercise 6 to randomly choose the computer's move from will need to be extended to add `"lizard"` and `"spock"`.

5. The computer reads your mind!

⚠Warning!⚠

This part is *really not easy*. You'll need to be familiar with **strings**, **loops**, **if statements**, **lists**, **tuples**, and **dictionaries**. If you've completed the rest of this booklet and you need help with this part, feel free to ask a tutor.

Your challenge, should you choose to accept it:

Extension 7: Add artificial intelligence to the computer

Having the computer throw moves randomly is a start, but it can be even smarter. By learning your patterns, the computer can predict what move you'll play next!

Your program can use a list to store the history of the player's moves, and a dictionary to store the statistics of patterns in those moves. Initialise these at the start of your code:

```
history = []
stats = {}
```

You need to make sure that these are at the start of your code, before and outside of the main loop.

At the end of each turn, the human's move should be added to the list `history`. Once there are 3 or more moves in the history, you should start collecting information in the dictionary `stats` so that it looks like this:

```
{
("scissors", "rock"): {"scissors": 4, "spock": 7},
("rock", "rock"): {"scissors": 2, "rock": 2, "paper": 8},
…
}
```

In this dictionary, the keys are a 2-tuple of the first two moves the human played, and the values are dictionaries containing the number of times the human followed those two moves with each possible other move. For example, to find which moves have followed the pair of moves `("scissors", "spock")`:

```
>> stats[("scissors", "spock")]
{"scissors": 4, "rock": 5}
```

Here we can see that `("scissors", "spock")` was followed by `"scissors"` 4 times, and followed by `"rock"` 5 times. The computer could then guess that the player is most likely to follow `("scissors", "spock")` with `"rock"`. And if the player is most likely to follow with `"rock"`, then the computer should use a move that defeats `"rock"` - either `"paper"` or `"spock"`.

Good luck!