Rock Paper Scissors – step by step

1. **Import Random**

Using the description and rules above, you can make a game of rock paper scissors. Before you dive in, you’re going to need to [import](https://realpython.com/python-import/) the module you’ll use to simulate the computer’s choices:

import random

Awesome! Now you’re able to use the different tools inside random to randomize the computer’s actions in the game. Now what? Since your users will also need to be able to choose their actions, the first logical thing you need is a way to take in user input.

1. **Take User Input**

[Taking input from a user](https://realpython.com/python-input-output/) is pretty straightforward in Python. The goal here is to ask the user what they would like to choose as an action and then assign that choice to a variable:

user\_action = input("Enter a choice (rock, paper, scissors): ")

This will prompt the user to enter a selection and save it to a variable for later use. Now that the user has selected an action, the computer needs to decide what to do.

1. **Make the Computer Choose**

A competitive game of rock paper scissors involves [strategy](https://arstechnica.com/science/2014/05/win-at-rock-paper-scissors-by-knowing-thy-opponent/). Rather than trying to develop a model for that, though, you can save yourself some time by having the computer select a random action. [Random selections](https://realpython.com/python-random/) are a great way to have the computer choose a [pseudorandom](https://en.wikipedia.org/wiki/Pseudorandom_number_generator) value.

You can use **random.choice()** to have the computer randomly select between the actions:

possible\_actions = ["rock", "paper", "scissors"]

computer\_action = random.choice(possible\_actions)

This allows a random element to be selected from the list. You can also [print](https://realpython.com/python-print/) the choices that the user and the computer made:

print(f"\nYou chose {user\_action}, computer chose {computer\_action}.\n")

Printing the user and computer actions can be helpful to the user, and it can also help you debug later on in case something isn’t quite right with the outcome.

1. **Determine a Winner**

Now that both players have made their choice, you just need a way to decide who wins. Using an if … elif … else block, you can compare players’ choices and determine a winner:

if user\_action == computer\_action:

print(f"Both players selected {user\_action}. It's a tie!")

elif user\_action == "rock":

if computer\_action == "scissors":

print("Rock smashes scissors! You win!")

else:

print("Paper covers rock! You lose.")

elif user\_action == "paper":

if computer\_action == "rock":

print("Paper covers rock! You win!")

else:

print("Scissors cuts paper! You lose.")

elif user\_action == "scissors":

if computer\_action == "paper":

print("Scissors cuts paper! You win!")

else:

print("Rock smashes scissors! You lose.")

By comparing the tie condition first, you get rid of quite a few cases. If you didn’t do that, then you’d need to check each possible action for user\_action and compare it against each possible action for computer\_action. By checking the tie condition first, you’re able to know what the computer chose with only two conditional checks of computer\_action.

And that’s it! All combined, your code should now look like this:

import random

user\_action = input("Enter a choice (rock, paper, scissors): ")

possible\_actions = ["rock", "paper", "scissors"]

computer\_action = random.choice(possible\_actions)

print(f"\nYou chose {user\_action}, computer chose {computer\_action}.\n")

if user\_action == computer\_action:

print(f"Both players selected {user\_action}. It's a tie!")

elif user\_action == "rock":

if computer\_action == "scissors":

print("Rock smashes scissors! You win!")

else:

print("Paper covers rock! You lose.")

elif user\_action == "paper":

if computer\_action == "rock":

print("Paper covers rock! You win!")

else:

print("Scissors cuts paper! You lose.")

elif user\_action == "scissors":

if computer\_action == "paper":

print("Scissors cuts paper! You win!")

else:

print("Rock smashes scissors! You lose.")

You’ve now written code to take in user input, select a random action for the computer, and decide the winner! But this only lets you play one game before the program finishes running.

1. **Play Several Games in a Row**

Although a single game of rock paper scissors is super fun, wouldn’t it be better if you could play several games in a row? **Loops** are a great way to create recurring events. In particular, you can use a [while loop](https://realpython.com/python-while-loop/) to play indefinitely:

import random

while True:

user\_action = input("Enter a choice (rock, paper, scissors): ")

possible\_actions = ["rock", "paper", "scissors"]

computer\_action = random.choice(possible\_actions)

print(f"\nYou chose {user\_action}, computer chose {computer\_action}.\n")

if user\_action == computer\_action:

print(f"Both players selected {user\_action}. It's a tie!")

elif user\_action == "rock":

if computer\_action == "scissors":

print("Rock smashes scissors! You win!")

else:

print("Paper covers rock! You lose.")

elif user\_action == "paper":

if computer\_action == "rock":

print("Paper covers rock! You win!")

else:

print("Scissors cuts paper! You lose.")

elif user\_action == "scissors":

if computer\_action == "paper":

print("Scissors cuts paper! You win!")

else:

print("Rock smashes scissors! You lose.")

play\_again = input("Play again? (y/n): ")

if play\_again.lower() != "y":

break

Notice the highlighted lines above. It’s important to check if the user wants to play again and to break if they don’t. Without that check, the user would be forced to play until they terminated the console using Ctrl+C or a similar method.

The check for playing again is a check against the string "y". But checking for something specific like this might make it harder for the user stop playing. What if the user types "yes" or "no"? String comparison is often tricky because you never know what the user might enter. They might do all lowercase, all uppercase, or even a mixture of the two.

Here are the results of a few different string comparisons:

>>> play\_again = "yes"

>>> play\_again == "n"

False

>>> play\_again != "y"

True

Hmm. That’s not what you want. The user might not be too happy if they enter "yes" expecting to play again but are kicked from the game.

1. **Describe an Action With enum.IntEnum**

Because string comparisons can cause problems like you saw above, it’s a good idea to avoid them whenever possible. One of the first things your program asks, however, is for the user to input a string! What if the user inputs "Rock" or "rOck" by mistake? Capitalization matters, so they won’t be equal:

>>> print("rock" == "Rock")

False

Since capitalization matters, "r" and "R" aren’t equal. One possible solution would be to use [numbers](https://realpython.com/python-numbers/) instead. Assigning each action a number could save you some trouble:

ROCK\_ACTION = 0

PAPER\_ACTION = 1

SCISSORS\_ACTION = 2

This allows you to reference different actions by their assigned number. Integers don’t suffer the same comparison issues as strings, so this could work. Now you can have the user input a number and compare it directly against those values:

user\_input = input("Enter a choice (rock[0], paper[1], scissors[2]): ")

user\_action = int(user\_input)

if user\_action == ROCK\_ACTION:

# Handle ROCK\_ACTION

Because input() returns a string, you need to [convert the return value to an integer](https://realpython.com/convert-python-string-to-int/) using int(). Then you can compare the input to each of the actions above. This works well, but it might rely on you naming variables correctly in order to keep track of them. A better way is to use **enum.IntEnum** and define your own action class!

Using enum.IntEnum allows you to create attributes and assign them values similar to those shown above. This helps clean up your code by grouping actions into their own [namespaces](https://realpython.com/python-namespaces-scope/) and making the code more expressive:

from enum import IntEnum

class Action(IntEnum):

Rock = 0

Paper = 1

Scissors = 2

This creates a custom Action that you can use to reference the different types of actions you support. It works by assigning each attribute within it to a value you specify.

Comparisons are still straightforward, and now they have a helpful class name associated with them:

>>> Action.Rock == Action.Rock

True

Because the member values are the same, the comparison is equal. The class names also make it more obvious that you want to compare two actions.

**Note:** To learn more about enum, check out the [Build Enumerations of Constants With Python’s Enum](https://realpython.com/python-enum/).

You can even create an Action from an int:

>>> Action.Rock == Action(0)

True

>>> Action(0)

<Action.Rock: 0>

Action looks at the value passed in and returns the appropriate Action. This is helpful because now you can take in the user input as an int and create an Action from it. No more worrying about spelling!

## **The Flow(chart) of Your Program**

Although rock paper scissors might seem uncomplicated, it’s important to carefully consider the steps involved in playing it so that you can be sure your program covers all possible scenarios. For any project, even small ones, it’s helpful to create a **flowchart** of the desired behavior and implement code around it. You could achieve a similar result using a bulleted list, but it’d be harder to capture things like loops and [conditional logic](https://realpython.com/python-conditional-statements/).

Flowcharts don’t have to be overly complicated or even use real code. Just describing the desired behavior ahead of time can help you fix problems before they happen!

Here’s a flowchart that describes a single game of rock paper scissors: 

Each player selects an action and then a winner is determined. This flowchart is accurate for a single game as you’ve coded it, but it’s not necessarily accurate for real-life games. In real life, the players select their actions simultaneously rather than one at a time like the flowchart suggests.

In the coded version, however, this works because the player’s choice is hidden from the computer, and the computer’s choice is hidden from the player. The two players can make their choices at different times without affecting the fairness of the game.

Flowcharts help you catch possible mistakes early on and also let you see if you want to add more functionality. For example, here’s a flowchart that describes how to play games repeatedly until the user decides to stop:



Without writing code, you can see that the first flowchart doesn’t have a way to play again. This approach allows you to tackle issues like these before programming, which helps you create neater, more manageable code!

## **Split Your Code Into Functions**

Now that you’ve outlined the flow of your program using a flowchart, you can try to organize your code so that it more closely resembles the steps you’ve identified. One natural way to do this is to [create a function](https://realpython.com/defining-your-own-python-function/) for each step in the flowchart. Functions are a great way to separate larger chunks of code into smaller, more manageable pieces.

You don’t necessarily need to create a function for the conditional check to play again, but you can if you’d like. You can start by importing random if you haven’t already and defining your Action class:

import random

from enum import IntEnum

class Action(IntEnum):

Rock = 0

Paper = 1

Scissors = 2

Hopefully this all looks familiar so far! Now here’s the code for get\_user\_selection(), which doesn’t take in any arguments and [returns](https://realpython.com/python-return-statement/) an Action:

def get\_user\_selection():

user\_input = input("Enter a choice (rock[0], paper[1], scissors[2]): ")

selection = int(user\_input)

action = Action(selection)

return action

Notice how you take in the user input as an int and get back an Action. That long message for the user is a bit cumbersome, though. What would happen if you wanted to add more actions? You’d have to add even more text to the prompt.

Instead, you can use a [list comprehension](https://realpython.com/list-comprehension-python/) to generate a portion of the input:

def get\_user\_selection():

choices = [f"{action.name}[{action.value}]" for action in Action]

choices\_str = ", ".join(choices)

selection = int(input(f"Enter a choice ({choices\_str}): "))

action = Action(selection)

return action

Now you no longer need to worry about adding or removing actions in the future! Testing this out, you can see how the code prompts the user and returns an action associated with the user’s input value:

>>> get\_user\_selection()

Enter a choice (rock[0], paper[1], scissors[2]): 0

<Action.Rock: 0>

Now you need a function for getting the computer’s action. Like get\_user\_selection(), this function should take no arguments and return an Action. Because the values for Action range from 0 to 2, you’re going to want to generate a random number within that range. **random.randint()** can help with that.

random.randint() returns a random value between a specified minimum and maximum (inclusive). You can use len() to help figure out what the upper bound should be in your code:

def get\_computer\_selection():

selection = random.randint(0, len(Action) - 1)

action = Action(selection)

return action

Because the Action values start counting from 0, and len() starts counting from 1, it’s important to do len(Action) - 1.

When you test this, there won’t be a prompt. It will simply return the action associated with the random number:

>>> get\_computer\_selection()

<Action.Scissors: 2>

Looking good! Next, you need a way to determine a winner. This function will take two arguments, the user’s action and the computer’s action. It doesn’t need to return anything since it’ll just display the result to the console:

def determine\_winner(user\_action, computer\_action):

if user\_action == computer\_action:

print(f"Both players selected {user\_action.name}. It's a tie!")

elif user\_action == Action.Rock:

if computer\_action == Action.Scissors:

print("Rock smashes scissors! You win!")

else:

print("Paper covers rock! You lose.")

elif user\_action == Action.Paper:

if computer\_action == Action.Rock:

print("Paper covers rock! You win!")

else:

print("Scissors cuts paper! You lose.")

elif user\_action == Action.Scissors:

if computer\_action == Action.Paper:

print("Scissors cuts paper! You win!")

else:

print("Rock smashes scissors! You lose.")

This is pretty similar to the first comparison you used to determine a winner. Now you can just directly compare Action types without worrying about those pesky strings!

You can even test this out by passing different options to determine\_winner() and seeing what gets printed:

>>> determine\_winner(Action.Rock, Action.Scissors)

Rock smashes scissors! You win!

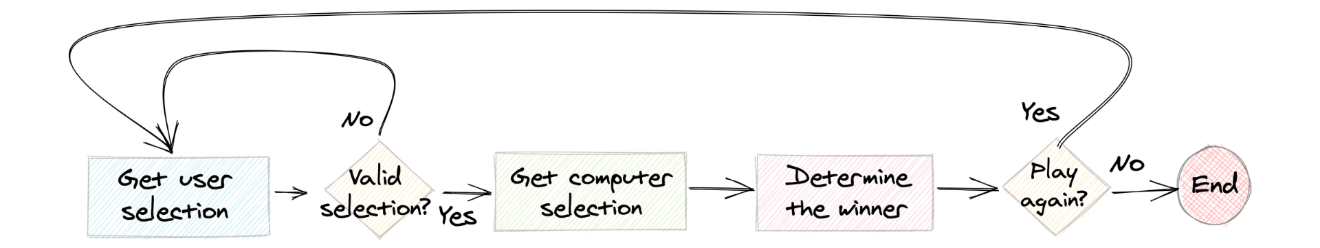
Since you’re creating an action from a number, what would happen if your user tried to create an action from 3? Remember, largest number you’ve defined so far is 2:

>>> Action(3)

ValueError: 3 is not a valid Action

Whoops! You don’t want that to happen. Where in the flowchart could you add some logic to ensure the user has entered a valid choice?

It makes sense to include the check immediately after the user has made their choice:



If the user enters an invalid value, then you repeat the step for getting the user’s choice. The only real requirement for the user selection is that it’s between 0 and 2, inclusive. If the user’s input is outside this range, then a ValueError exception will be raised. To avoid displaying the default error message to the user, you can **handle** the exception.

**Note:** Exceptions can be tricky! For more information, check out [Python Exceptions: An Introduction](https://realpython.com/python-exceptions/).

Now that you’ve defined a few functions that reflect the steps in your flowchart, your game logic is a lot more organized and compact. This is all your while loop needs to contain now:

while True:

try:

user\_action = get\_user\_selection()

except ValueError as e:

range\_str = f"[0, {len(Action) - 1}]"

print(f"Invalid selection. Enter a value in range {range\_str}")

continue

computer\_action = get\_computer\_selection()

determine\_winner(user\_action, computer\_action)

play\_again = input("Play again? (y/n): ")

if play\_again.lower() != "y":

break

Doesn’t that look a lot cleaner? Notice how if the user fails to select a valid range, then you use continue rather than break. This makes the code continue to the next iteration of the loop rather than break out of it.