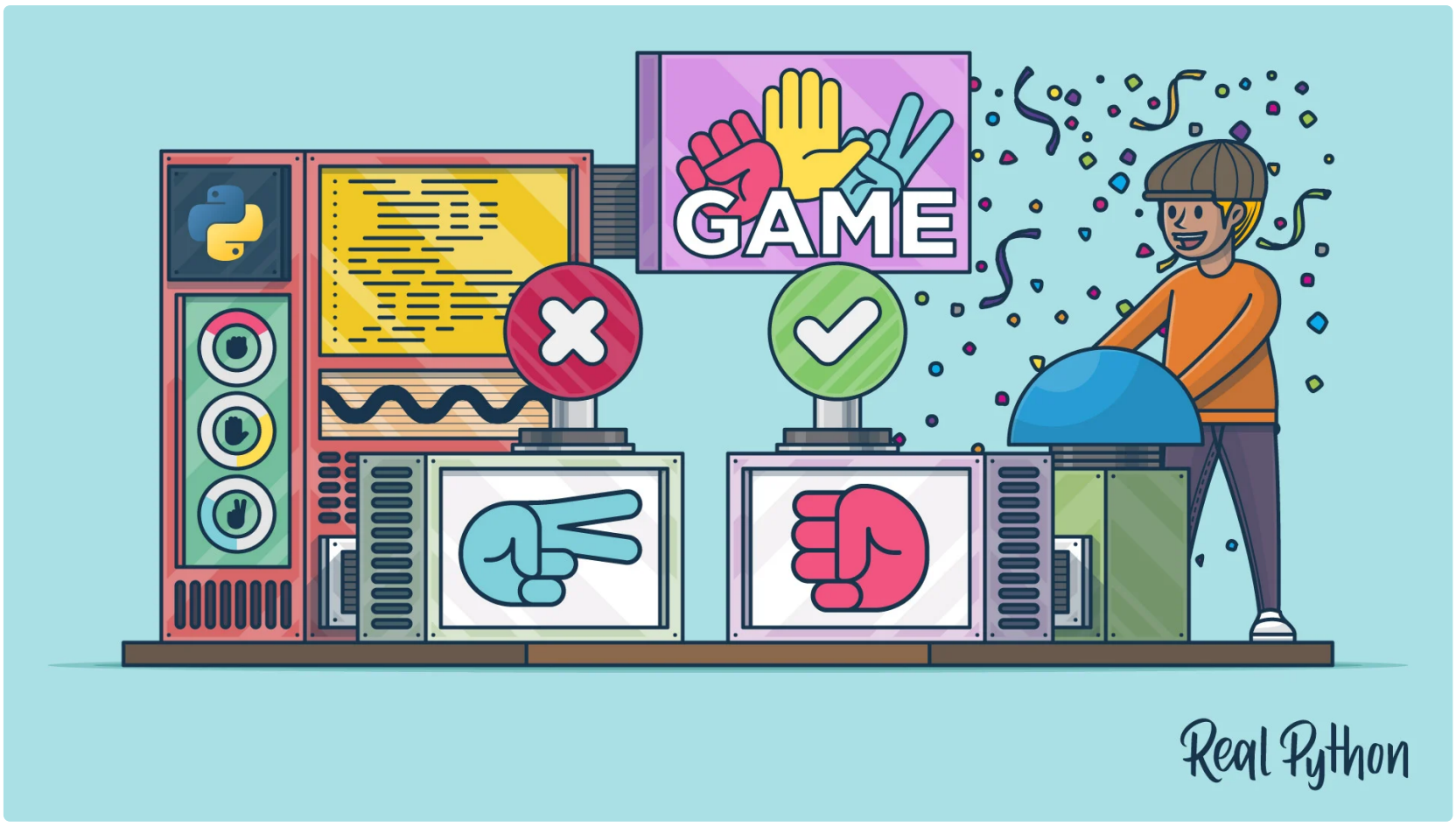


Make Your First Python Game: Rock, Paper, Scissors!

by Chris Wilkerson 6 Comments basics gamedev python

Mark as Completed

Tweet Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Rock, Paper, Scissors With Python: A Command Line Game**

Game programming is a great way to learn how to program. You use many tools that you'll see in the real world, plus you get to play a game to test your results! An ideal game to start your Python game programming journey is rock paper scissors.

In this tutorial, you'll learn how to:

Improve Your Python

- Code your own **rock paper scissors** game
- Take in user input with `input()`
- Play several games in a row using a `while` **loop**
- Clean up your code with `Enum` and **functions**
- Define more complex rules with a **dictionary**

What Is Rock Paper Sc

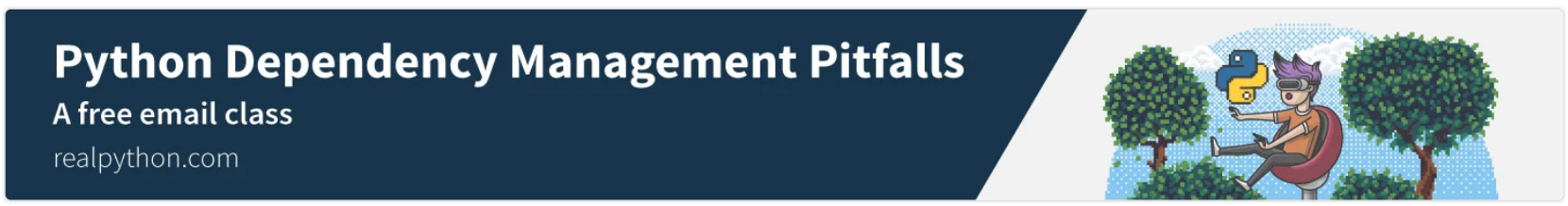
You may have played rock paper scissors choice of players for a team.

If you're unfamiliar, rock paper scissors is scissors" and then simultaneously form t downward), or a pair of scissors (two fing

- **Rock** smashes scissors.
- **Paper** covers rock.
- **Scissors** cut paper.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Now that you have the rules down, you c



Play a Single Game of Rock Paper Scissors in Python

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 the module you'll use to simulate the computer's choices:

Python

```
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.

Take User Input

Taking input from a user 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:

Python

```
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.

Make the Computer Choose

A competitive game of rock paper scissors involves strategy. 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 are a great way to have the computer choose a pseudorandom value.

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

Python

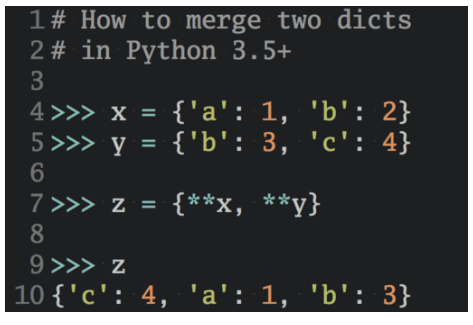
```
possible_actions = ["rock", "paper", "scissors"]
computer_action = random.choice(possible_actions)
```

This allows a random element to be selected from the list. You can also print the choices that the user and the computer made:

Python

```
print(f"\nYou chose {user_action}, cc
```

Printing the user and computer actions c
something isn't quite right with the outcc



Determine a Winner

Now that both players have made their cl
block, you can compare players' choices

Python

```
if user_action == computer_action:
    print(f"Both players selected {us
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:

Python

```
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 {com
```

```
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! Y
elif user_action == "scissors":
    if computer_action == "paper":
        print("Scissors cuts paper! Y
    else:
        print("Rock smashes scissors!
```

You've now written code to take in user ir
this only lets you play one game before th

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 to play indefinitely:

Python

```
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.

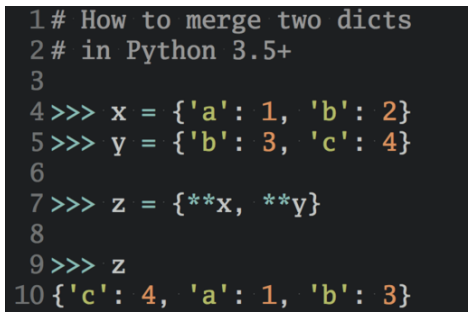
The check for playing again is a check against the string `"y"`. But checking for something specific like this might make it harder for the user stop playing. What if the user types `"yes"` or `"no"`? String comparison is often tricky because you never know what the user might enter. They might do all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

Here are the results of a few different string comparisons:

Python

```
>>> play_again = "yes"
>>> play_again == "n"
False
>>> play_again != "y"
True
```

Hmm. That's not what you want. The use
kicked from the game.



Describe an Action Wit

Because string comparisons can cause pr
possible. One of the first things your prog
`"Rock"` or `"rOck"` by mistake? Capitalizatic

Python

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

Since capitalization matters, `"r"` and `"R"` aren't equal. One possible solution would be to use numbers instead. Assigning each action a number could save you some trouble:

Python

```
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:

Python

```
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 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 and making the code more expressive:

Python

```
from enum import IntEnum

class Action(IntEnum):
    Rock = 0
    Paper = 1
    Scissors = 2
```

This creates a custom `Action` that you can use to reference t
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:

Python

```
>>> 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

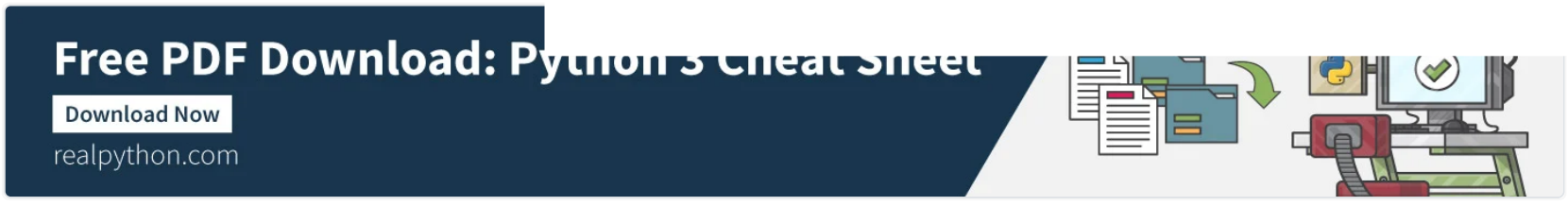
You can even create an `Action` from an `in`

Python

```
>>> Action.Rock == Action(0)
True
>>> Action(0)
<Action.Rock: 0>
```

`Action` looks at the value passed in and re
the user input as an `int` and create an `Act`

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

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.

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 Fu

Now that you've outlined the flow of you closely resembles the steps you've identi flowchart. Functions are a great way to se

You don't necessarily need to create a fur can start by importing `random` if you have

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Python

```
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 an `Action`:

Python

```
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 to generate a portion of the input:

Python

```
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:

Python >>>

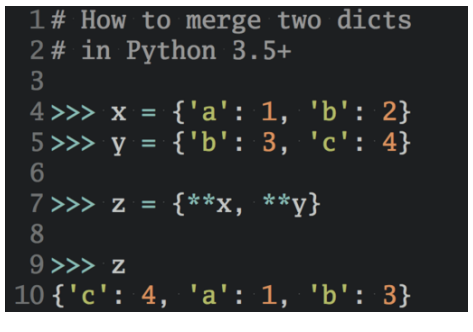
```
>>> get_user_selection()
Enter a choice (rock[0], paper[1], scissors[2]): 0
<Action.Rock: 0>
```

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:

Python

```
def get_computer_selection():
    selection = random.randint(0, ler
    action = Action(selection)
    return action
```



Because the `Action` values start counting

When you test this, there won't be a pron

Python

```
>>> 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:

Python

```
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:

Python >>>

```
>>> 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`:

Python >>>

```
>>> Action(3)
ValueError: 3 is not a valid Action
```


Whoops! You don't want that to happen. Where in the flowc

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 yo
the user selection is that it's between `0` ar
exception will be raised. To avoid display

> **Note:** Exceptions can be tricky! For mo

Now that you've defined a few functions
organized and compact. This is all your `wh`

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

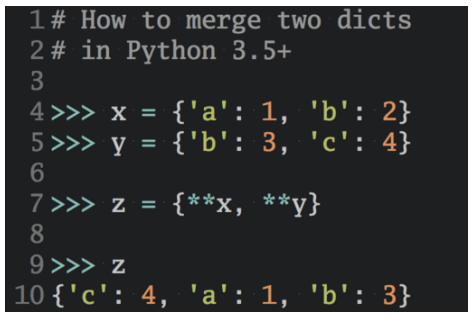

Python

```
while True:
    try:
        user_action = get_user_select
    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.

Rock Paper Scissors … Lizard Spock

If you've seen *The Big Bang Theory*, then you may be familiar with rock paper scissors lizard Spock. If not, then here's a diagram depicting the game and the rules deciding the winner:



You can use the same tools you learned a
and create values for lizard and Spock. Th
`get_computer_selection()` to incorporate
work.

Instead of adding a lot of `if` … `elif` … `else` statements to your code, you can use a dictionary to help show the relationships between actions. Dictionaries are a great way to show a **key-value relationship**. In this case, the **key** can be an action, like scissors, and the **value** can be a list of actions that it beats.

So what would this look like for your `determine_winner()` with only three options? Well, each `Action` can beat only one other `Action`, so the list would contain only a single item. Here's what your code looked like before:

Python

```
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.")
```

Now, instead of comparing to each `Action`, you can have a dictionary that describes the victory conditions:

Python

```
def determine_winner(user_action, computer_action):
    victories = {
        Action.Rock: [Action.Scissors],  # Rock beats scissors
        Action.Paper: [Action.Rock],  # Paper beats rock
        Action.Scissors: [Action.Paper]  # Scissors beats paper
    }

    defeats = victories[user_action]
    if user_action == computer_action:
        print(f"Both players selected {user_action.name}. It's a tie!")
    elif computer_action in defeats:
        print(f"{user_action.name} be
    else:
        print(f"{computer_action.name
```


You still do the same as before and check
compare the action that the `user_input` b
use the membership operator `in` to check

Since you no longer use long `if` … `elif` ..
You can start by adding lizard and Spock

Python

```
class Action(IntEnum):
    Rock = 0
    Paper = 1
    Scissors = 2
    Lizard = 3
    Spock = 4
```

Next, add all the victory relationships from the diagram. Make sure to do this underneath `Action` so that `victories` will be able to reference everything in `Action`:

Python

```
victories = {
    Action.Scissors: [Action.Lizard, Action.Paper],
    Action.Paper: [Action.Spock, Action.Rock],
    Action.Rock: [Action.Lizard, Action.Scissors],
    Action.Lizard: [Action.Spock, Action.Paper],
    Action.Spock: [Action.Scissors, Action.Rock]
}
```

Notice how now each `Action` has a list containing two elements that it beats. In the basic rock paper scissors implementation, there was only one element.

Because you intentionally wrote `get_user_selection()` to accommodate new actions, you don't have to change anything about that code. The same is true for `get_computer_selection()`. Since the length of `Action` has changed, the range of random numbers will also change.

Look at how much shorter and more manageable the code is now! To see the full code for your complete program, expand the box below.

Rock Paper Scissors Lizard Spock Code Show/Hide

That's it! You've implemented rock paper scissors lizard Spock in Python code. Double-check to make sure you didn't miss anything and give it a playthrough.

Conclusion

Congratulations! You just finished your first Python game! You now know how to create rock paper scissors from scratch, and you're able to expand the number of possible actions in your game with minimal effort.

In this tutorial, you learned how to:

- Code your own **rock paper scissors** game
- Take in user input with `input()`
- Play several games in a row using a `while` **loop**
- Clean up your code with `Enum` and **functions**
- Describe more complex rules with a **dictionary**

These tools will continue to help you throughout your many programming adventures. If you have any questions, then feel free to reach out in the commer

Ma

Watch Now This tutorial has a relate written tutorial to deepen your unders

Get a short & sweet **Python Trick** de days. No spam ever. Unsubscribe an team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


About **Chris Wilkerson**

Chris is an avid Pythonista and writes for Real Python.

» More about Chris

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

David

Geir Arne

Joanna

What Do You Think?

Rate this article:

Tweet Share Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: basics gamedev python

Recommended Video Course: Rock, Paper, Scissors With Python: A Command Line Game

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

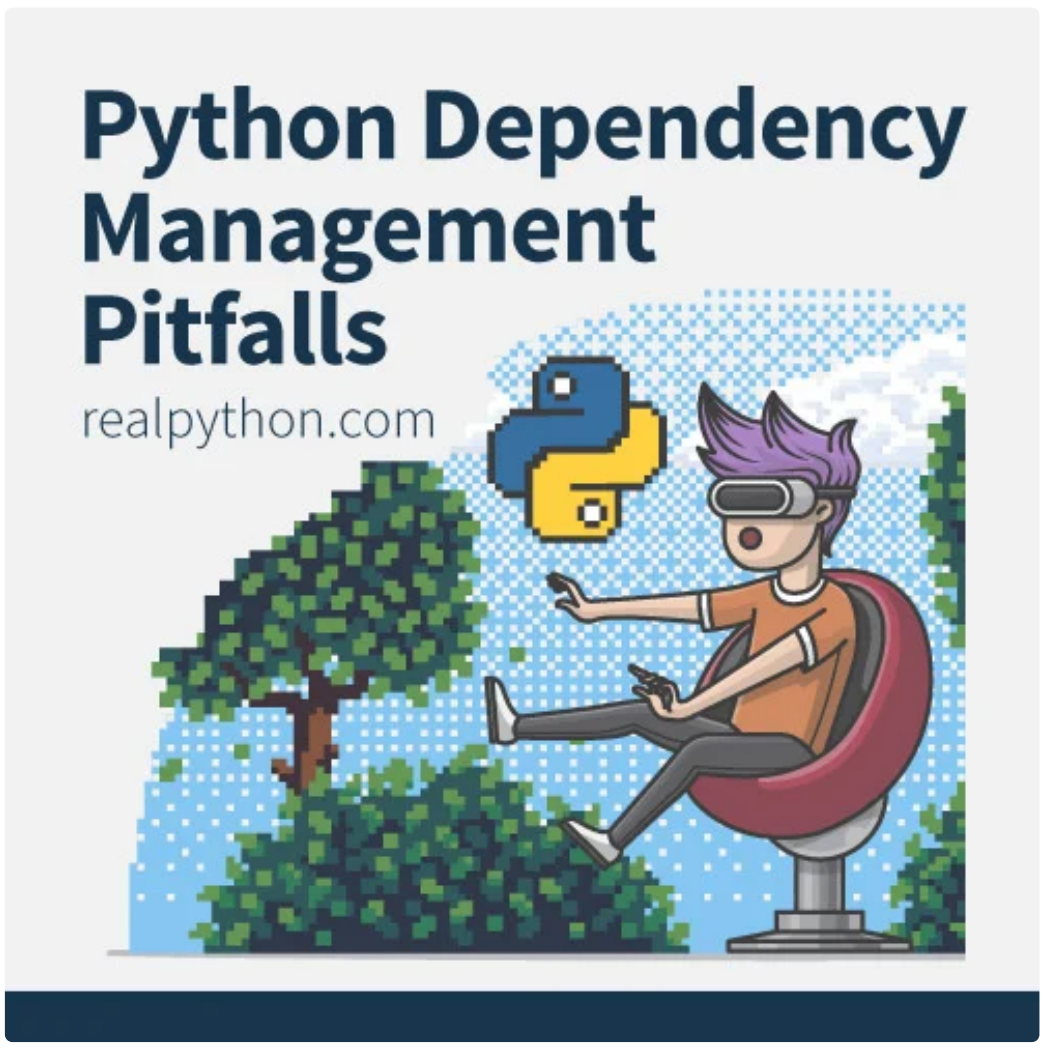


Table of Contents

Mark as Completed