Some Personal Philosophies

One-Liners Suck

```
def is_winner(player, opponent):
  # this could be written as one long combined if statement.
  # I personally prefer to write lots of small pieces of logic.
  # I find this gives me more places to inspect and debug the program,
  # especially as programs and logic become more complex.
  if player == "rock" and opponent == "scissors":
    return True
  elif player == "paper" and opponent == "rock":
    return True
  elif player == "scissors" and opponent == "paper":
    return True

  # if they didn't win they didn't win!
  return False
```

Avoid Tiny Variable Names

It won't be hard to find programming advice telling you to use descriptive variable names and match the stylistic naming conventions of projects and programming languages. You should use `UPPER_CASE` and `camelCase` and `kebab-case` and `snake_case` where appropriate.

- Java and JavaScript prefer `camelCase`.
- Ruby and Python prefer `snake_case`.
- HTML and CSS prefer `kebab-case`.
- `UPPER_CASE` is often reserved for global variables or class constants.

It's a good idea to use descriptive variable names. If you're writing a program to compare American and Canadian gas prices you'll need variables to keep track of the exchange rate between Looneys and Bucks, and to represent the ratio of liters to gallons.

In the example below it's a much better idea to take the time to write out long, monotonous, descriptive variable names. Some programmers may be tempted to use (as an extreme) single-letter variable names, but it makes the logic of the program hard to follow, debug, and decipher by your future self and other programmers.

The single-letter character name is an extreme example. It's (I hope obviously!) ridiculous to use single-letter names to keep track of things like different conversion ratios. Here's another non-obvious example where I've found it feels natural to use single-letter variable names, but using names even a little longer provides a nice benefit.

I prefer to use `xx` and `yy` instead of just `x` and `y` when dealing with coordinates!

The single-letter `x` and `y` names are totally legitimate semantically (they mean what they say, as opposed to `a` to store "Canadian dollar per liter"). The problem I find with them is they become hard to search for.

In the arbitrary code-snippet below consider what would happen if you wanted to search through code looking for things accessing the x and y coordinates. If the code used single-letter `x` and `y` variable names searching for "x" or "y" all sorts of false matches would come up matching single letters in other variable names and inside comments: "xray", "experimentFortyTwo", "arbitrary".

It's a small thing, but seriously I go out of my way to avoid single-letter variable names a lot!

Hard-to-Search Variable Names in Coordinates

```
class Point {
  constructor(xx, yy) {
    this.xx = xx
    this.yy = yy
  }
}

let xray = new Point(11, 22)
let experimentFortyTwo = new Point(42, 42)

// arbitrarily double the position of both
xray.xx *= 2
xray.xx *= 2
experimentFortyTwo.xx *= 2
experimentFortyTwo.yy *= 2
```

Useless Variable Names in Conversions

Some example information:

- 1 liter of gas costs $1.39 CAD
- 1 gallon of gas costs $3.50 USD
- 1 Canadian Dollar equals 0.75 United States Dollar (2019-11-21)
- 1 liter is 0.264172 gallons

```
a = 1.39
b = 3.50
c = .75
d = .264172
z = a / c / d
```

```
cad_per_liter = 1.39
dollar_per_gallon = 3.50
cad_per_dollar = .75
gallon_per_liter = .264172

canadian_gas_per_gallon_in_cad = cad_per_liter / gallon_per_liter
canadian_gas_per_gallon_in_usd = cad_per_liter / gallon_per_liter / cad_per_dollar

american_gas_per_gallon_in_cad = dollar_per_gallon / cad_per_dollar
american_gas_per_gallon_in_usd = dollar_per_gallon

print("one gallon in canada CAD", canadian_gas_per_gallon_in_cad)
print("one gallon in canada USD", canadian_gas_per_gallon_in_usd)

print("one gallon in america CAD", american_gas_per_gallon_in_cad)
print("one gallon in america USD", american_gas_per_gallon_in_usd)
```

Zero to One, One to Two, Two to Many

- Going from zero to one is hard
- Going from one to two is hard
- Going from two to three is hard
- Going from three to fourth, and on, is easy

There's a significant amount of work to go from zero to one, building something for the first time.

Serialization: Saving the State of the Game

At some point you'll want to save the state of your game. Maybe you'll want to save their current game to file so they can play later. Maybe you'll want to save the state of the game so you can make an AI to explore making different moves. Maybe you'll want to save the current state of the game across a few turns to prevent players from getting stuck in a loop (like "Ko" in Go).

Ko

Serialization is the process of taking the state of the game and writing it down in a way that you can load the game back exactly as it was.

If you serialized a game of Tic Tac Toe you should write down what the board looks like, and write down which player should take the next turn.

Serializing a game of poker is more complex. Here's pieces of state you'll want to save:

- The cards in each players's hand
- The current amount of money each player has
- Write down who's turn it is

Saving the state of other parts of the game can become even more complex. Do you want to be able to save the game in the middle of a hand, in the middle of a betting round, in the middle of while players can still discard cards?

Should the state of the deck and discarded cards be saved, or should you reshuffle the deck when the game is loaded? If the deck is reshuffled each time the game is loaded players can cheat the system by reloading the game over and over until they draw cards they want.

- Write down which players have discarded and redrawn cards.
- Write down where you are in the betting process.
- Write down what sort of side pots you've established, if any.

Thinking about how you will serialize and load the game back from serialized state can help you think about how you should structure the internal state of the game initially.

Shallow Copy vs Deep Copy

JavaScript const .push array

I Won't Say There's A Right Way, but I Will Tell You Different Ramifications