CHICAS
centre for health informatics, computing, and statistics

# Week 10 - Python Engineering Software

Barry Rowlingson

Lancaster University
Medical School

**For this talk I want to say a few more things about software design, specifically for more complex systems and programs using objects and classes.**

# Rock-Paper-Scissors

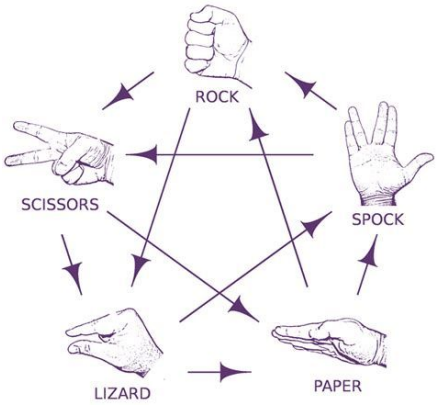
CHICAS centre for health informatics, computing, and statistics

Paper wraps Rock

Scissors cuts Paper

Rock blunts Scissors

Random play is ⅓ win, ⅓ lose, ⅓ draw




Here's the setup. Rock-paper-scissors. Two players, each chooses simultaneously one of Rock, Paper, or Scissors and the winner is decided. Rock blunts scissors, scissors cut paper, and paper wraps rock. If the players choose the same the round is a draw. Random play results in equal numbers of wins, loses, and draws. There's also various extensions such as "rock, paper, scissors, lizard, spock" with five choices. The game dates back to at least the 1600s in China and is also known as Roshambo.

# Task

CHICAS centre for health informatics, computing, and statistics

Write a program to play multiple games of RPS against a variety of possible opponents.

We want to write some code to play multiple games of RPS, using a variety of possible strategies. Now we could hack out a single long python script to do this, with a loop at its core, but it pays to think about things first. How do we address the problem at hand?

# Think about "Actors"

CHICAS centre for health informatics, computing, and statistics

- The "mechanic" of the game itself
  - Game start
  - Rounds
  - Scoring
  - Game end
  - Winner
- The players
  - Make a play
  - See the round outcome

One strategy is to think about what the separate "actors" are in the system, and how they interact. We can break the game down into the "mechanic" or process of the game itself, and the players. The "game" needs to keep score, get the players' choices, announce the winner etc. The players need to make plays and they also get to see the result of each round as it happens.

# Utility Functions

CHICAS centre for health informatics, computing, and statistics

Who wins, given two choices?

- Check valid and invalid options!
- Write tests

```
def win(p1, p2):
    """ test for wins and draws. Return 0 for a draw,
        or number of winning player (1 or 2).

>>> win("Rock","Paper")
2
>>> win("Rock","Rock")
0
>>> win("Scissors","Rock")
2
>>> win("Rock","Scissors")
1
>>> win("Rock", "AAAARGH")
1
>>> win("AAARGH!", "OUCH!")
0
>>> win("OUCH!", "Paper")
2

"""
```

Cover all the cases!

There's nearly always a few simple utility functions that we can write early on and test thoroughly so we're not debugging these in the middle of debugging the game code. This set of tests is a good start at making sure my "win" detection function is correct. Notice it handles invalid input, resulting in a lose for the invalid input unless there's two invalid inputs in which case its a draw. These tests are written using the Python "doctest" system which means they can be run and tested from the command line.

# Players

CHICAS centre for health informatics, computing, and statistics

```
class Player():
    def __init__(self, position):
        """ new player at position """
        pass
    def start(self, nrounds):
        """ called at start of a new game """
        pass
    def play(self, round):
        """ return play for this round """
        pass
    def played(self, p1, p2, scores):
        """ called after a round played """
        pass
```

Now we can think about the players. We'll need a class so we can create player objects. That means we need the standard Python "__init__" method, and we probably want to create the player with the position (player 1 or 2) set. Then to start a game the game will call the "start" method to set the player up for a new game, then call the players "play" method to get one of the plays. To tell the player what happened there's a "played" method which will tell the player what players 1 and 2 did and the current score. This then is a first draft. We might adjust these methods, or add new ones, at some point.

# Random Player

CHICAS centre for health informatics, computing, and statistics

```
PLAYS = ["Rock","Paper","Scissors"]

class Player():
    def __init__(self, position):
        self.position = position
    def start(self, nrounds):
        self.nrounds = nrounds
    def play(self, round):
        return random.sample(PLAYS, 1)[0]
    def played(self, p1, p2, scores):
        pass
```

```
>>> import rockpaperscissors
>>> p1 = rockpaperscissors.Player(10)
>>> p1.play(1)
'Rock'
>>> p1.play(2)
'Scissors'
>>> p1.play(3)
'Paper'
```

Our first player is a random one, so we can write the play method to choose one of the valid plays uniformly. As written this is all we need since this player doesn't need to keep track of the score or the round, it just returns one of the plays. Nevertheless I'm storing the input values in the object anyway for some of the other methods. But now I have a player I can test from the command line. By importing the file and creating a new instance of Player, I can invoke the "play" method and see that I get back a play.

# Game Logic

CHICAS centre for health informatics, computing, and statistics

```
class Game():
    def __init__(self, nrounds, player1, player2):
        self.nrounds = nrounds
        self.player1 = player1
        self.player2 = player2
        self.scores = [0,0,0] # draw, p1win, p2win
```

The game logic now comes into consideration. We'll create a new Game class to hold the "state" of the game. An initial state for a game consists of knowing how many rounds, and who the two players are, so we'll pass those to the constructor and store them.

# Game Logic

CHICAS centre for health informatics, computing, and statistics

```
class Game():
    def __init__(self, nrounds, player1, player2):

    def play(self):
        self.player1.start(self.nrounds)
        self.player2.start(self.nrounds)

        for i in range(self.nrounds):
            p1 = self.player1.play(i)
            p2 = self.player2.play(i)
            winner = win(p1, p2)
            self.scores[winner] += 1
            self.player1.played(p1, p2, self.scores)
            self.player2.played(p1, p2, self.scores)

        return self.scores
```

Then we can play the game. We'll tell the players that we've started, then loop over the rounds. For each round we ask the players for their play, compute the winner, then add one to the winner's score. Then we call the "played" method to let the players know what happened, in case they are smart players that want to react to opponents plays…

# Play!

CHICAS centre for health informatics, computing, and statistics

```
def main(nrounds):
    p1 = Player(1)
    p2 = Player(2)
    g = Game(nrounds, p1, p2)
    return g.play()
```

```
>>> import rockpaperscissors
>>> rockpaperscissors.main(3000)
[988, 1034, 978]
```

Then here's a function that I can import, it sets up two player objects, and plays, returning the scores. I can import this and run a number of games, and I should get the ⅓, ⅓, ⅓ outcomes. Looks good.

# Command Line

CHICAS centre for health informatics, computing, and statistics

```
if __name__ == "__main__":
    import sys
    nrounds = int(sys.argv[1])
    print(main(nrounds))
```

```
$ python rockpaperscissors.py 300000
[99618, 100103, 100279]
```

I can also do the standar python __name__ trick so I can run this from the command line.

# Human Player

```
class HumanPlayer(Player):
    def __init__(self, position):
        super().__init__(position)
    def play(self, round):
        choose = input("Rock/Paper/Scissors? ")
        return choose

def main(nrounds):
    p1 = HumanPlayer(1)
    p2 = Player(2)
    g = Game(nrounds, p1, p2)
    return g.play()
```

```
$ python rockpaperscissors.py 3
Rock/Paper/Scissors? Rock
Rock/Paper/Scissors? Rock
Rock/Paper/Scissors? Scissors
[1, 1, 1]
```

Playing two random players against each other is a bit dull. What do we need for a human player? The main thing is to get an input, and we can do that with the Python “input” function. So I make a new HumanPlayer class, and I could make this inherit from the existing player class, and overwrite the “play” method to ask for an input. I can change “p1” in the main function to “HumanPlayer” and now I can play against a random number generator…

# Better Output

CHICAS centre for health informatics, computing, and statistics

```
Rock/Paper/Scissors? Rock
p1='Rock' p2='Rock' score is [1, 0, 0]
Rock/Paper/Scissors? Paper
p1='Paper' p2='Rock' score is [1, 1, 0]
Rock/Paper/Scissors? Scissors
p1='Scissors' p2='Scissors' score is [2, 1, 0]
Rock/Paper/Scissors? Rock
p1='Rock' p2='Rock' score is [3, 1, 0]
Rock/Paper/Scissors? Scissors
p1='Scissors' p2='Paper' score is [3, 2, 0]
[3, 2, 0]
```

That output is a bit sparse, so a few more lines of code later I've got a method that is a bit more explanatory.

# A Smarter Computer

CHICAS centre for health informatics, computing, and statistics

```
class SmartPlayer(Player):
    def __init__(self, position):
        super().__init__(position)
    def start(self, nrounds):
        self.opposition_plays = [0, 0, 0]
    def play(self, round):
        """ choose the play that would beat the play
        that the opposition has played the most """
        raise NotImplementedError
    def played(self, p1, p2, scores):
        """ increment correct element in self.opposition_plays"""
        raise NotImplementedError
```

Human players are not random, and they are generally beatable. How would we make a smart computer player? Here's an outline for a player that counts the plays made by the opposition and then chooses the play that would most likely beat the opposition. At the start of a game it sets a list to zeroes to count the number of rock, paper, and scissors plays, and then in the "played" method it would add one to whatever the opponent played on that round. The "play" method would then choose the best response based on previous plays. Even smarter systems could look for sequences, runs, or patterns in the opposition plays to perform even better against non-random humans.

To test this you could also write a "BadHumanPlayer" class that returned non-random plays - maybe it always played "Rock", or strictly alternated "Rock", "Paper", "Scissors" in that order, and you could test various coded "SmartPlayers" against those strategies.

## Thoughts

CHICAS centre for health informatics, computing, and statistics

- Break a complex system down
- Spot the independent components
- Think about "actors" and how they interact
- Make things testable

- Learn from version 1 and don't fear scrapping it and starting again!

That's very much an overview of a system design process. Break it down, work out the components and how they interact, then write model examples and testable units. Another big lesson is to learn. What I've shown here is not a first version and not a final version. The first Player class didn't have all the methods and I ran into problems implementing the more complex player types, so I revised everything. As shown I can see one important revision that needs doing - I should take most of the behaviour out of the "Player" class and make a "RandomPlayer" class so that both it and the HumanPlayer class inherit from Player, so I can keep common "Player" behaviour coded there rather than duplicated.