Trading Cards

When you set out to collect trading cards, be it Pokémon or Panini, the aim is the same: get every single card in the set. Exactly how difficult is that for a lone collector? What if, as the name suggests, you trade with a collaborator or five? How much more efficient can you be if you work together?

How it works

A finite set of cards / stickers / collectibles is available to purchase. Your aim is to own one of each item. The catch is, when you purchase new items, you have no way of knowing which one you are getting. If you end up with a card that's a duplicate of one you already have, that's not much use to you. That's where trading comes in handy.

I work alone…

If you have the cash, and the patience, there's no reason why you couldn't collect all the cards you need on your own. And when you first begin, you feel like you're doing pretty well. After all, the very first card you buy is definitely not yet in your collection. And, chances are, neither are the second or third cards. But the more you buy, the more of the cards you already have, so the fewer you actually need. The chance of a new card being an unseen card drop with each new card you add to your collection. For instance, if you have already collected 635 of the 636 Panini football stickers you need, the chance of your next sticker being the one you need is $\frac{1}{636}$. That means that, on average, you'll have to buy 636 more stickers to complete your collection.

Stronger together…

If you have a friend who is also collecting, then you have an advantage. Any time you receive a duplicate card, if they don't yet have it, you can trade it with them. If you're feeling really generous, you can simply give them any cards they need that you don't, and hope for the same from them. On average, you should both complete your collections more quickly.

Try it with dice

A quick way to get a feel for this is to roll a normal six-sided die repeatedly, trying to hit all six numbers at least once each. If you really can't be bothered to get up and open your dice safe (every home should have one), why not write a quick bit of code to do it for you? Before you start, try to make a prediction about how long it is likely to take, and maybe jot down some pseudo-code or a rough specification here (you'll be surprised how helpful it is, even for a quick little program, to clarify your aims and process first).

Building a quick simulation with Python

First things first

The essentials: I want to simulate rolling a fair, six-sided die repeatedly until I have seen at least one of every number from 1 to 6, and I want to print out how many rolls it took.
Nice to have: A way of doing this repeatedly, showing me a summary of the results for a large number of simulations, maybe even on a graph.

What will I need?

- The **random** module: using **random.randint(1, 6)** I can generate my dice rolls.
- An accumulator like **num_rolls** to keep track of the number of rolls I make.
- Some way of keeping track of which numbers I still need to collect. I could make this a list, like **[1, 2, 3, 4, 5, 6]**, from which I remove numbers as I roll them.

```
1   import random
2
3   def stats(data):
4       data.sort()
5       n = len(data)
6       lo = data[0]
7       lq = data[n // 4]
8       me = data[n // 2]
9       uq = data[3 * n // 4]
10      hi = data[-1]
11      return lo, lq, me, uq, hi
12
13  results = []
14  trials = 10000
15  for i in range(trials):
16      to_collect = [i for i in range(1, 7)]
17      rolls = 0
18      while len(to_collect) > 0:
19          rolls += 1
20          dice = random.randint(1, 6)
21          if dice in to_collect:
22              to_collect.remove(dice)
23      results.append(rolls)
24
25  print(stats(results))
```

We import the **random** module so we can get our dice rolls on line 20.

The **stats** function sorts the data provided, and returns the five key statistics you would use to draw a box-plot. They give a decent overview of the data and the approximate shape of the distribution without going OTT.

Creating an empty **results** list lets me save the number of rolls required each time (see line 23). List comprehensions helps me make numbers from 1 to 6 (line 16), and I run a **while** loop, rolling dice and only removing values from my list when I roll them.
At the end, I run my **stats** function and show the headline figures.

On average, it takes around 13 rolls, but 25% of the time more than 18.
*See if you can adapt the **stats** function to give you the 95th percentile.*

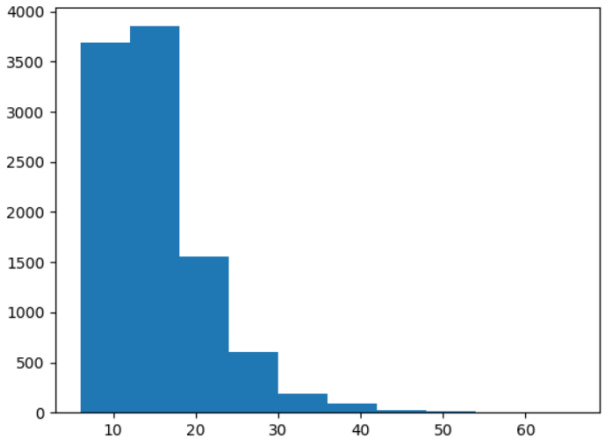
A bit of maths

If we call N_k the average number of rolls you would expect to make to see a new number after you've already seen k of them, then let's consider N_4: If we've already seen four of the six numbers, we have a $\frac{2}{6}$ chance of getting a new one on our very next roll, and a $\frac{4}{6}$ chance of needing… $1 + N_4$ rolls to get a new one (the roll we just had that gave us a duplicate, plus the average number of rolls required to get a new number!
This means that $N_4 = \frac{2}{6}(1) + \frac{4}{6}(1 + N_4)$ which gives $N_4 = \frac{6}{2} = 3$ In general, $N_k = \frac{6}{6-k}$ and the total average number of rolls is: $N_0 + N_1 + \cdots + N_6 = \frac{6}{6} + \frac{6}{5} + \cdots + \frac{6}{1} = 14.7$.

What about a graph?

The **matplotlib.pyplot** module will let us draw histograms without too much effort:

```
2    import matplotlib.pyplot as plt
...
26   print(stats(results))
27   plt.hist(results)
28   plt.savefig("Dice.png")
```

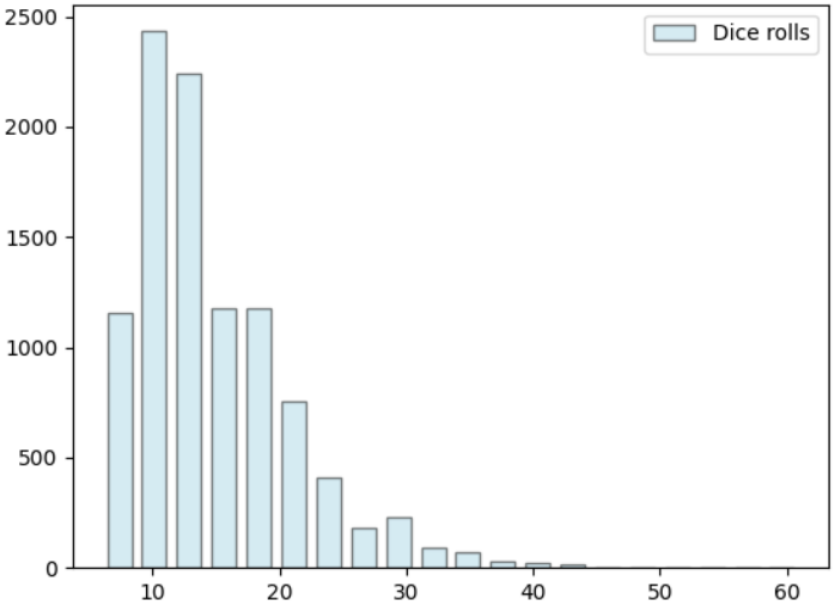


Remember to import **matplotlib.pyplot** at the start, then you can simply invoke **hist** to create your graph. If you're working with Python in a normal environment, you can use **plt.show()**, but in repl.it you will need **plt.savefig** to generate the graphic.

Improving your graph

The **hist** object allows for quite a lot of customisation. Here are a few of the optional keyword arguments you can try out (more details in the documentation here):

```
plt.hist(results, bins=20, alpha = 0.5, color = "lightblue", edgecolor = "black", label = "Dice rolls", rwidth = 0.7)
plt.legend()
plt.savefig("Dice.png")
```



bins refers to the number of categories the data will be sorted into.
alpha gives transparency (useful if you have more than one data set sharing the axes)
color is the fill colour of the bars, and **edgecolor** the colour of the border.
label does nothing unless you also run **plt.legend()** to show the key.
rwidth allows you to add spacing to your histogram. Technically, there should be spacing if the data is discrete, but not if it's continuous. But in reality, it's up to you – far worse sins of data representation are committed on national media every day…

More than one graph?

If you want more than one histogram shown on the same set of axes, just run **plt.hist** again with the second set of data (use **alpha** so they don't obscure each other)

```
plt.hist(results1, alpha=0.5)
plt.hist(results2, alpha=0.5)
plt.savefig("Overlapping histograms.png")
```

Where were we?

Getting back to our card collection, we can pretty easily adapt our code for a larger collection, and even plot a range of results on the same chart.

```
1   import random
2   import matplotlib.pyplot as plt
3
4   def sim(n):
5     results = []
6     trials = 100
7     for i in range(trials):
8       to_collect = [i for i in range(n)]
9       rolls = 0
10      while len(to_collect) > 0:
11        rolls += 1
12        dice = random.randint(0, n - 1)
13        if dice in to_collect:
14          to_collect.remove(dice)
15      results.append(rolls)
16    return results
17
18  for n in [20, 50, 100]:
19    results = sim(n)
20    plt.hist(results, bins=20, alpha = 0.5, edgecolor = "black", label = f"{n} cards", rwidth = 0.7)
21  plt.legend()
22  plt.savefig("Dice.png")
```

I've put our code into its own function so I can run it repeatedly with different numbers of cards. The random number generator now spits out a number between 0 and 19 if n is 20, etc.

The results are then plotted using the same **hist** function as before. Note that when we use this line over and over, each set of data is plotted on the same axes (which is fine for this particular set of values).

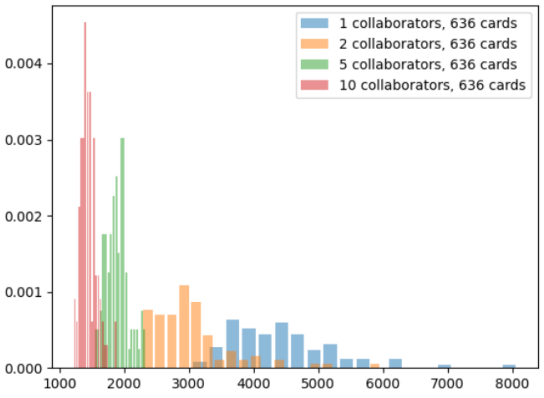
I get by with a little help from my friends

Let's see how we can bring in collaborators now. The simplest way would be to imagine, say, three friends all working together to fill three albums. So for a collection of just 4 cards, you'd want a list like this: **[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]**. The items will get removed in the same way, and of course it will take longer than completing one album, but you get to share the cost with your friends.

```
1   import random
2   import matplotlib.pyplot as plt
3
4   def sim(n, collaborators):
5     results = []
6     trials = 100
7     for i in range(trials):
8       to_collect = [i for i in range(n)] * collaborators
9       rolls = 0
10      while len(to_collect) > 0:
11        rolls += 1
12        dice = random.randint(0, n - 1)
13        if dice in to_collect:
14          to_collect.remove(dice)
15      results.append(rolls // collaborators)
16    return results
17
18  num_cards = 636
19  for i in [1, 2, 5, 10]:
20    plt.hist(sim(num_cards, i), bins=20, alpha=0.5, rwidth=0.8,
      density=True, label=f"{i} collaborators, {num_cards} cards")
21    print(f"Done for {i} collaborators.")
22  plt.legend()
23  plt.savefig("Results.png")
24  print("Done")
```

The **sim** function now accepts the additional argument **collaborators**, which I use on line 8 to make the list longer (note that **[1, 2, 3] * 2** in Python gives **[1, 2, 3, 1, 2, 3]**).

The loop at the end runs the simulation for a single collector, 2 collaborators, 5 and then 10. In each loop, the results are plotted on the histogram.



What's next?

You could use Python classes to define individuals if you want a more realistic trading scenario than everyone-shares-everything, keeping cards collected in a dictionary, and trading only when others have something you need.