Making Money

Once we can conceptualize an object in Python, we can start creating populations of objects – virtual people, animals or less animate objects. Then we can have them interact to investigate the effects of applying simple rules to the long term behaviour of the group. A classic example is trade and wealth.

Money is:

- ***A shared illusion, of value to individuals because of its value to other individuals, who value it because of its value to other individuals…***
- ***A bargain with the future. I make sacrifices today in order to make tomorrow better. Money is the guarantee of that better future.***

The beginning of illusion…

Money facilitates trade: You catch fish, I make weapons and Bob is a dental surgeon. I need a tooth pulled, but Bob doesn't need any more weaponry. You need a spear, but I don't like fish. If we're lucky, Bob will need some fish, and we can complete the chain. Chances are, Bob doesn't like fish, and we'll have to get ever more creative. If only there were some universally valued commodity...

Imagine you conned us all into thinking that a certain rare yellow shiny metal was worth having. You give me gold for a spear, I give Bob gold for an extraction, and he uses the gold to buy a pair of forceps from some guy who offers it to you in exchange for some fish. At this point, even if you didn't really believe in the value of gold initially, the fact that everyone else believes in it makes it worth having. Not everyone wants kippers, but they all seem to like that weird yellow stuff. It has come to represent fish and spears, dental work and forceps. Plus, it won't go bad in a few days like fish, and it's a lot easier to carry around than a clutch of pole-arms.

From imaginary value to real value…

The really curious thing about money is that it creates wealth – real, tangible, value.

The next logical step once you have efficient trade is specialization – each individual doing what they do best, and reaping the benefits of scaling and efficiency rather than trying to do everything themselves and doing none of it very well.

A family has 10 tokens, pillaged from a board game. The children are given a token every time they help out around the house (clearing the table, doing the laundry), and they can give their parents tokens to do things for them (making a hot chocolate, helping tidy their room). The result is that tokens are exchanged, and make their way between family members in a seemingly pointless way, but along the way *things get done*. The dishes get washed, the lawn gets mowed, a batch of cookies gets baked. All the work that takes place enriches the entire family, and the ten little tokens that make their way from person to person are what makes it all possible. Not only that, but different people have different preferences, and if you're vacuuming your bedroom, it's not much extra effort to vacuum the whole of the upstairs. Making cookies? Bake enough for everyone.

Creating the simulation

Trade creates wealth, seemingly out of nothing, because every transaction (in theory) enriches both parties involved (otherwise they wouldn't engage in the transaction). However, it is rarely the case that both parties benefit equally. Someone may well be getting a great deal, and the other maybe not so good. Maybe you pay over the odds for something that turned out to not be as good as you thought. Maybe you don't have enough money to buy in bulk, so you pay more per item than those who are better off. At the extreme ends, it is easier to get a loan on good terms if you are in a strong financial position, and much, much harder if you will struggle to pay it off.

We can simulate this by creating a population where each person starts with a random amount of money, and each individual randomly interacts with other individuals, usually to their mutual benefit, but perhaps skewed in favour of the more wealthy.
Note that, for simplicity, we will treat all 'wealth' as cash. For instance:

Alice has £30	**Bob has £50**
• Alice instigates a trade with Bob (maybe Alice fixes his car, which enriches him, and Bob pays Alice for her work, which enriches her). • Let the total wealth generated by the transaction be determined by a random number between 0 and 8 (a tenth of their total wealth – the richer people are, the more lucrative their average transactions will be). • Pick a random number between 0 and 80 (their total wealth). • If it's between 0 and 30 (Alice's wealth), then she gets the majority of the benefit (let's say 80% of the total value), and if it's not, he gets the 80% share.	

What would happen to a large population of people who interact in this way? Notice that the element of randomness introduced works in favour of those with the most money, but not all transactions will favour the most wealthy, since there is an element of randomness.

Let's simulate it with Python

Step 1: Create a Person object, which will have the capabilities we want

```
1   import random
2
3   class Person:
4     def __init__(self):
5       self.initial = random.randint(10, 1000)
6       self.cash = self.initial
7
8     def trade_with(self, other):
9       total_wealth = self.cash + other.cash
10      reward = random.randint(0, total_wealth // 10)
11      if random.randint(0, total_wealth) < self.cash:
12        self.cash += int(0.8 * reward)
13        other.cash += int(0.2 * reward)
14      else:
15        self.cash += int(0.2 * reward)
16        other.cash += int(0.8 * reward)
17
18    def report(self):
19      print(f"From £{self.initial} to £{self.cash}")
```

(We import the **random** module first).

The class definition comes next, starting with the **__init__** method, defining the starting values. In addition to the actual amount of money, I'm keeping track of what each person starts with.

The **trade_with** method is an interaction with another **Person** object, so I can invoke things like **other.cash**.
The **reward** depends on the cash each person has, and the person who gains the most is likely (but not certain) to be the one with the most cash.

The **report** function is to give us information about the person.

Step 2: Build a population, and make them trade with each other!

```
22  num_people = int(input("How many people should we create? "))
23  num_trades = int(input("How many trades should we simulate? "))
```

If we allow the user to determine the most commonly changing values (the number of people and the number of trades), we can run the program repeatedly with different parameters without messing about with the code.

```
24  population = []
25  for i in range(num_people):
26      new_person = Person()
27      population.append(new_person)
28
```

A list called **population** is created, to hold all the **Person** objects we're about to create. Then we make new people one at a time, and append them to the list.

```
29  for i in range(num_trades):
30      a, b = random.sample(population, 2)
31      a.trade_with(b)
```

The next step is to make people trade, at random. The simplest way to do this is to use the **random.sample** function to pick out two random people from the population. I've called them **a** and **b**, but these labels are just temporary – each time the loop iterates, a fresh sample is chosen, and those two people will engage in trade.

Step 3: Extract information

```
33  money = [p.cash for p in population]
```

This first line uses something called ***list comprehensions***. It is essentially a neater way of performing common tasks with lists. Line 33 is completely equivalent to:

```
money = []
for p in population:
    money.append(p.cash)
```

The notation should be familiar to mathematicians, since it borrows heavily from set notation: $E = \{2k : k \in \mathbb{N}\}$ defines the set of even numbers, E, which is a collection of elements of the form $2k$ where k is a member of the set $\mathbb{N}$, the natural numbers.

```
34  richest = money.index(max(money))
35  poorest = money.index(min(money))
36  population[richest].report()
37  population[poorest].report()
```

By using the **index** method for lists, we can identify the *position* of the largest element and that of the smallest. Since these positions are equivalent to the population, we can then extract information directly from those people.

```
39  money.sort()
40  top_20 = int(100 * sum(money[int(num_people * 0.8):]) / sum(money))
41  print(f"The richest 20% of people own {top_20}% of the wealth")
```

The last 3 lines shown above use the **sort** method of lists to put **money** in ascending order. By *slicing* the list we can extract just the top 20% of values: **money[10:]** would produce a new list containing all elements of **money** from item number 10 to the end. We have used **num_people** to start our sublist from a point 80% of the way through. The rest converts the amounts to a percentage of the total, which can be printed out.

What's next?

There are a few easy tweaks you can make that would add functionality to your program, and the possibilities are endless when it comes to factoring in the more complex elements of even a fairly basic financial system:

- Wrap the code that builds a population, makes them trade and reports back in a **while True** loop, so once you've completed one simulation, the program resets, allowing you to modify parameters and try another set of values immediately.
- Can you **run a series of simulations**, and analyse the results to determine the most likely outcome for different starting values?
- Can you **count the number of trades** that take place, and identify the average gain from each trade?
- Make your own adjustments to the **trade_with** method. Maybe set it so that rich people don't trade with poor people at all, or make it fairer by changing the 80% payoff structure, or the biased decision-maker.
- Add in some unforeseen random **financial disasters** – alongside the trading that happens in the loop around line 30, you could randomly select someone to lose £100 (not a big deal for the wealthy, but possibly disastrous for the poor), or 50% of their money (somewhat worse for the rich).
- On the flip side, what happens if a random individual (perhaps someone with money to burn) randomly **gives money** to another individual)?
- Build in a hefty **tax** on the rich, or a cap on earnings for the super-rich.
- **Let them steal** – maybe build in the potential for someone to steal from someone else. This could be a random element within the **trade_with** method, or built in to the initial attributes of a Person. Maybe a risk is associated with attempted theft: get caught and risk a fine, but if you're desperate enough, or if your trading partner is rich enough, it might be tempting.
- Investigate the **Pareto distribution** (20% of a population possess 80% of the wealth at any one time). Does this hold true for your simulations? What happens as the number of trades you simulate increases?
- Can you construct a graph to show the various fortunes of the individuals? Rather than simply graphing their cash over time (which may rapidly get huge and not reveal very much), maybe you can graph the relative rankings of each individual to see if the people at the top stay at the top, or if there is any fluctuation over time.
- How about making every trade a **zero-sum game**? What if certain individuals have a propensity to cheat, while others are more likely to cooperate for the greater good? A simple example might be: if both parties cooperate, they each gain £20. If one cheats and the other cooperates, the cheat gains £30 and the co-operator loses £10. If, however, both cheat, nobody gains anything.
- Model a **pyramid scheme** or Ponzi scheme. A single individual recruits people who pay a subscription to their 'boss' for the privilege of joining the scheme. They then attempt to recruit others, recouping their own subscription fee by earning a cut of the subscription fee of their own recruits (and passing a portion of it up the chain to their own boss in turn). If it costs you £100 to join the scheme, and you recruit 3 people each for £100, you keep three lots of £50 and pass on the other £150 to your boss. Not only that, but anyone they recruit will earn you £50 (of which you keep £25, and pass £25 on to your boss).