

Quantitative Methods 2

Ollie Ballinger

10/10/2022

Table of contents

Welcome

Welcome to BASC0005 - Quantitative Methods: Data Science and Visualisation

This course teaches quantitative skills, with an emphasis on the context and use of data. Students learn to focus on datasets which will allow them to explore questions in society – in arts, humanities, sports, criminal justice, economics, inequality, or policy. Students are expected to work with Python to carry out data manipulation (cleaning and segmentation), analysis (for example, deriving descriptive statistics) and visualisation (graphing, mapping and other forms of visualisation). They will engage with literatures around a topic and connect their datasets and analyses to explore and decide wider arguments, and link their results to these contextual considerations. Below is an outline of the course:

Week	Lecture	Theme
1	Introduction	Question
2	Data	Data Acquisition
3	Spatial Data	Data Acquisition
4	Text Data	Data Acquisition
5	Bias	Exploration
6	Group Pitches	Group Pitches
7	Distributions	Exploration
8	Regression	Modeling
9	Machine Learning	Modeling
10	Visualization	Visualization

1 Python Recap

1.1 Workshop 1



1.2 Using Python

In this course, we'll make extensive use of *Python*, a programming language used widely in scientific computing and on the web. We will be using Python as a way to manipulate, plot and analyse data. This isn't a course about learning Python, it's about working with data - but we'll learning a little bit of programming along the way.

By now, you should have done the prerequisites for the module, and understand a bit about how Python is structured, what different commands do, and so on - this is a bit of a refresher to remind you of what we need at the beginning of term.

The particular flavour of Python we're using is *iPython*, which, as we've seen, allows us to combine text, code, images, equations and figures in a *Notebook*. This is a *cell*, written in *markdown* - a way of writing nice text. Contrast this with *code* cell, which executes a bit of Python:

```
print(1+1)
```

2

The Notebook format allows you to engage in what Don Knuth describes as [Literate Programming](#):

[...] Instead of writing code containing documentation, the literate programmer writes documentation containing code. No longer does the English commentary injected into a program have to be hidden in comment delimiters at the top of the file, or under procedure headings, or at the end of lines. Instead, it is wrenched into the daylight and made the main focus. The “program” then becomes primarily a document directed at humans, with the code being herded between “code delimiters” from where it can be extracted and shuffled out sideways to the language system by literate programming tools. [Ross Williams](#)

1.3 Libraries

We will work with a number of *libraries*, which provide additional functions and techniques to help us to carry out our tasks.

These include:

Pandas: we'll use this a lot to slice and dice data

matplotlib: this is our basic graphing software, and we'll also use it for mapping

nltk: The Natural Language Tool Kit will help us work with text

We aren't doing all this to learn to program. We could spend a whole term learning how to use Python and never look at any data, maps, graphs, or visualisations. But we do need to understand a few basics to use Python for working with data. So let's revisit a few concepts that you should have covered in your prerequisites.

1.4 Variables

Python can broadly be divided in verbs and nouns: things which *do* things, and things which *are* things. In Python, the verbs can be *commands*, *functions*, or *methods*. We won't worry too much about the distinction here - suffice it to say, they are the parts of code which manipulate data, calculate values, or show things on the screen.

The simplest proper noun object in Python is the *variable*. Variables are given names and store information. This can be, for example, numeric, text, or boolean (true/false). These are all statements setting up variables:

```
n = 1
```

```
t = "hi"
```

```
b = True
```

Now let's try this in code:

```
n = 1
t = "hi"
b = True
```

Note that each command is on a new line; other than that, the *syntax* of Python should be fairly clear. We're setting these variables equal to the letters and numbers and phrases and booleans. **What's a boolean?**

The value of this is we now have values tied to these variables - so every time we want to use it, we can refer to the variable:

```
n
```

```
1
```

```
t
```

```
'hi'
```

```
b
```

```
True
```

Because we've defined these variables in the early part of the notebook, we can use them later on.

*Advanced: where do **classes** fit into this noun/verb picture of variables and commands?*

1.5 Where is my data?

When we work in excel and text editors, we're used to seeing the data onscreen - and if we manipulate the data in some way (averaging or summing up), we see both the inputs and outputs on screen. The big difference in working with Python is that we don't see our variables all of the time, or the effect we're having on them. They're there in the background, but it's usually worth checking in on them from time to time, to see whether our processes are doing what we think they're doing.

This is pretty easy to do - we can just type the variable name, or "print(*variable name*)":

```
n = n+1
print(n)
print(t)
print(b)
```

```
2
```

```
hi
```

```
True
```

1.6 Flow

Python, in common with all programming languages, executes commands in a sequence - we might refer to this as the “ineluctable march of the machines”, but it’s more common referred to as the *flow* of the code (we’ll use the word “code” a lot - it just means commands written in the programming language). In most cases, code just executes in the order it’s written. This is true within each *cell* (each block of text in the notebook), and it’s true when we execute the cells in order; that’s why we can refer back to the variables we defined earlier:

```
print(n)
```

2

If we make a change to one of these variables, say n:

```
n = 3
```

and execute the above “print n” command, you’ll see that it has changed n to 3. So if we go out of order, the obvious flow of the code is confused. For this reason, try to write your code so it executes in order, one cell at a time. At least for the moment, this will make it easier to follow the logic of what you’re doing to data.

Advanced: what happens to this flow when you write *functions* to automate common tasks?

Exercise - Setting up variables:

1. Create a new cell.
2. Create the variables “name”, and assign your name to it.
3. Create a variable “Python” and assign a score out of 10 to how much you like Python.
4. Create a variable “prior” and if you’ve used Python before, assign True; otherwise assign False to the variable
5. Print these out to the screen

1.7 Downloading Data

Lets fetch the data we will be using for this session. There are two ways in which you can upload data to the Colab notebook. You can use the following code to upload a CSV or similar data file.


```
from google.colab import files
uploaded = files.upload()
```

ModuleNotFoundError: No module named 'google.colab'

Or you can use the following cell to fetch the data directly from the QM2 server.

Let's create a folder that we can store all our data for this session

```
!mkdir data

!mkdir ./data/wk1
!curl https://s3.eu-west-2.amazonaws.com/qm2/wk1/data.csv -o ./data/wk1/data.csv
!curl https://s3.eu-west-2.amazonaws.com/qm2/wk1/sample_group.csv -o ./data/wk1/sample_gro
```

% Total		% Received		% Xferd		Average Speed		Time	Time	Time	Current
						Dload	Upload	Total	Spent	Left	Speed
100	203	100	203	0	0	527	0	--:--:--	--:--:--	--:--:--	527
% Total		% Received		% Xferd		Average Speed		Time	Time	Time	Current
						Dload	Upload	Total	Spent	Left	Speed
100	297	100	297	0	0	838	0	--:--:--	--:--:--	--:--:--	836

1.8 Storing and importing data

Typically, data we look at won't be just one number, or one bit of text. Python has a lot of different ways of dealing with a bunch of numbers: for example, a list of values is called a **list**:

```
listy = [1,2,3,6,9]
print(listy)
```

```
[1, 2, 3, 6, 9]
```

A set of values *linked* to an index (or key) is called a **dictionary**; for example:

```
dicty = {'Bob': 1.2, 'Mike': 1.2, 'Coop': 1.1, 'Maddy': 1.3, 'Giant': 2.1}
print(dicty)
```

```
{'Bob': 1.2, 'Mike': 1.2, 'Coop': 1.1, 'Maddy': 1.3, 'Giant': 2.1}
```

Notice that the list uses square brackets with values separated by commas, and the dict uses curly brackets with pairs separated by commas, and colons (:) to link a *key* (index or address) with a value.

(You might notice that they haven't printed out in the order you entered them)

***Advanced:** Print out 1) The third element of **listy**, and 2) The element of **dicty** relating to Giant

We'll discuss different ways of organising data again soon, but for now we'll look at *dataframes* - the way our data-friendly *library Pandas* works with data. We'll be using Pandas a lot this term, so it's good to get started with it early.

Let's start by importing pandas. We'll also import another library, but we're not going to worry about that too much at the moment.

If you see a warning about 'Building Font Cache' don't worry - this is normal.

```
import pandas

import matplotlib
%matplotlib inline
```

Let's import a simple dataset and show it in pandas. We'll use a pre-prepared ".csv" file, which needs to be in the same folder as our code.

```
data = pandas.read_csv('./data/wk1/data.csv')
data.head()
```

	Name	First Appearance	Approx height	Gender	Law Enforcement
0	Bob	1.2	6.0	Male	False
1	Mike	1.2	5.5	Male	False
2	Coop	1.1	6.0	Male	True
3	Maddy	1.3	5.5	Female	False
4	Giant	2.1	7.5	Male	False

What we've done here is read in a .csv file into a dataframe, the object pandas uses to work with data, and one that has lots of methods for slicing and dicing data, as we will see over the coming weeks. The head() command tells iPython to show the first few columns/rows of the data, so we can start to get a sense of what the data looks like and what sort of type of objects it represents.

2 Supplementary: Kaggle exercises

If you've gotten this far, congratulations! To further hone your skills, try working your way through the five [intro to programming notebooks on Kaggle](#). These cover a range of skills that we'll be using throughout the term. Kaggle is a very useful resource for learning data science, so making an account may not be a bad idea!

3 Intro to Pandas

3.1 Workshop 2 Open in Colab

In this workshop, our aim is to get used to working with more complex data that we've imported from external files. We'll start to graph it, and to slice and dice it, to select the bits we're interested in.

We will work with *pandas* to manipulate the data, and to derive measures and graphs that tell us a bit more than what the source data files tell us.

3.1.1 Aims

- Learn to import data to python using pandas
- Learn how access specific rows, columns and cells
- Plot the data
- Tidy up graphs to include axes

3.2 Introduction

We are going to work with some UK income data. The income data is packaged as a .csv file. The Pandas package knows how to handle this and put the data in a DataFrame, as we've seen. Let's examine the data and start to see what we can say about it. First of all, we have to find data - I'm interested in looking in data with a wide spread, so I looked for data on income in the UK.

This data is collected by the Office for National Statistics(ONS) : <http://www.ons.gov.uk/ons/datasets-and-tables/index.html?pageSize=50&sortBy=none&sortDirection=none&newquery=income+percentile> - but the exact data I want to see, income by percentile, is tricky to find.

I ended up using data from 2011, generated from a study called the Family Resources Survey and collated and tweaked by an independent research unit called the Institute of Fiscal Studies (IFS). The “tweaking” they do tends to be around the size of the family unit, and other factors which create economies of scale - hence they “equivalise” it. The IFS is quoted in UK Government documents, so we can have some trust in their impartiality, or at least accuracy

- of course, if we were publishing research about this, that's not really good enough and we'd want to reproduce, or at least understand and critique, their methodology rather than just trusting it!

e.g.:

<http://www.ifs.org.uk/wheredoyoufitin/about.php>

<https://en.wikipedia.org/wiki/Equivalisation>

3.3 Downloading the Data

Let's grab our income data from our course website and save it into our data folder. If you've not already created a data folder then do so using the following command. Don't worry if it generates an error, that means you've already got a data folder.

```
!mkdir data
```

mkdir: data: File exists

```
!mkdir data/wk2
```

```
!curl https://s3.eu-west-2.amazonaws.com/qm2/wk2/incomes.csv -o ./data/wk2/incomes.csv
```

mkdir: data/wk2: File exists

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload Upload	Total	Spent	Left	Speed
100 15154	100 15154	0 0	135k	0	--:--:--	--:--:--	143k

```
import pandas
import pylab
import matplotlib.pyplot as plt
# make the plots a little wider by default
%matplotlib inline
plt.style.use('ggplot')

pylab.rcParams['figure.figsize'] = (10., 8.)
```

```
data_path = "./data/wk2/incomes.csv"
```

```
income = pandas.read_csv(data_path, index_col=0)
```

```
income.head()
```

Percentile Point	Net equivalised household income in 2010-11, week	Childless couple, annual income	Couple, two children under 14
1	33.50	1,746.92	2,445.15
2	98.60	5,141.01	7,197.45
3	128.56	6,703.11	9,384.15
4	151.05	7,875.75	11,021.45
5	166.32	8,671.91	12,145.15

This is a simple dataframe - we see the percentile and an income. Note that I've told pandas to use the first column (the Percentile) as the index to make life easier.

The percentile tells us how people on that income rank - so the final category, 99% (which is really binned, so $99\% < n \leq 100\%$), is telling us how much "the 1%" earn. Let's find out:

```
income.tail()
```

Percentile Point	Net equivalised household income in 2010-11, week	Childless couple, annual income	Couple, two children under 14
95	1075.73	56,088.56	78,541.15
96	1174.48	61,237.18	85,731.15
97	1302.74	67,925.07	95,091.15
98	1523.31	79,425.23	111,141.15
99	2090.35	108,990.74	152,541.15

Well, they we have it - the 1% earn, on average, about £2000 a week. How does that compare to people in the 90% decile? We can access particular *rows* in a dataframe using `.loc[row index]`; because our index is the percentile point, we can just read it off:

```
income.loc[90]
```

Net equivalised household income in 2010-11, week	845.54
Childless couple, annual income	44,086.54
Couple, two children under 14	61,721.15
Couple, three children under 14	70,538.46
Couple with one child under 14	52,903.85
Couple with two children aged 15 to 18	73,183.65
Couple, two children under 14 plus dependent adult	76,269.71

```

Single adult                29,537.98
Lone parent, one child under 14  38,355.29
Lone parent, two children under 14  47,172.60
Lone parent, two children aged 15-18  58,635.10
ANNOTATIONS                NaN
1979 to 1996-97            2.50%
1996-97 to 2009-10         1.70%
1996-97 to 2010-11        1.20%
Name: 90, dtype: object

```

We can also select a range of values with the “colon” notation. This will select the 90-95th percentiles, for example:

```
income.loc[90:95]
```

Percentile Point	Net equivalised household income in 2010-11, week	Childless couple, annual income	Couple with children, annual income
90	845.54	44,086.54	61,721.54
91	876.63	45,707.74	63,991.54
92	911.29	47,514.54	66,521.54
93	957.14	49,905.23	69,801.54
94	1016.37	52,993.38	74,191.54
95	1075.73	56,088.56	78,521.54

3.4 Accessing parts of a dataframe

If we want to extract the actual value instead of just the whole row, we need to reference the *column* as well as the row. In pandas, columns are referenced by **column name**:

```
income['Net equivalised household income in 2010-11, week']
```

```

Percentile Point
1      33.50
2      98.60
3     128.56
4     151.05
5     166.32
...
95    1075.73

```

```
96    1174.48
97    1302.74
98    1523.31
99    2090.35
```

Name: Net equivalised household income in 2010-11, week, Length: 99, dtype: float64

So, to access a particular cell, we tell Python the row and the column (this is pretty simple - the same way we tell excel to access cell “A34” meaning Column A, Row 34). One way we do that in pandas is to select the column, and then use `.loc[]` on the index.

```
income['Net equivalised household income in 2010-11, week'].loc[90]
```

845.54

We’ve accessed row 90 of the column called ‘Net equivalised household income in 2010-11, week’; can we access the data the other way around - can we first take the row and then specify a column? Let’s try:

```
income.loc[90]['Net equivalised household income in 2010-11, week']
```

845.54

Yes, this seems to be working fine.

3.4.1 Extension

The reason for this is that selecting the column spits out a smaller dataframe, and all dataframes use “loc”, so we can use that. Another way to do this would be to use an explicit variable for the dataframe, along the lines of:

```
smallDataFrame = income['Net equivalised household income in 2010-11, week']
smallDataFrame.loc[90]
```

by doing income

```
['Net equivalised household income in 2010-11, week'].loc[90]
```

we’re taking the “smallDataFrame” object as an implicit (or hidden) output

If we want to look at a few rows of data, we can use a range:

```
income['Net equivalised household income in 2010-11, week'].loc[90:95]
```


Percentile Point

90	845.54
91	876.63
92	911.29
93	957.14
94	1016.37
95	1075.73

Name: Net equivalised household income in 2010-11, week, dtype: float64

So, to recap, we can now access a particular **row** using `loc[index number]`, a particular **column** with the square brackets formalism `dataframename['column name']`, or both `dataframename['column name'].loc[index number]`. We've made a start at being able to get to the bits of data we need.

3.5 Exercise:

How do the equivalised incomes of single adults and childless couples compare? Look at the 1st, 99th and 50th percentile and summarise what this tells you about the value or price of coupling.

3.6 Examining the Distribution

Returning to the overall statistics, the 90% percentile earns less than half the top percentile ("the 1%"); if you're taking home over £800 as a household, you're in the top 10% of earners.

How does 1. The income of "the 1%" compare with the mean and median across the population, as a proportion? 2. How does the 1% compare with the 90th percentile (the 10%)? 3. How does the 10% compare with the median and mean?

The 1% earn about 60 times the poorest groups in society - and we've made other comparisons. But that's not the whole story. Let's look at the income graph.

In pandas, we can plot this fairly easily...

```
income['Net equivalised household income in 2010-11, week'].plot()
plt.title('UK Net Equivalised Income by Percentile per week, 2010-11')
plt.xlabel('Income Percentile')
plt.ylabel('Income (Net, Equivalised) [GBP]')
```

```
Text(0, 0.5, 'Income (Net, Equivalised) [GBP]')
```



We see a curve that is pretty linear in the middle region, but curves rapidly upwards in the higher percentile and looks more like a power law.

3.6.1 Exercise: Means

Where does the mean appear here? Draw in a horizontal line to show the mean using **axhline**. Show the median on the same graph. What is the meaning of the median in this context?

Hint: Recall that last time we used *axvline* to highlight the mean and standard deviation by drawing vertical lines on the axis. Here, we use *axhline* to draw horizontal lines.

3.6.2 Extension: Accessing cells

There are a number of ways to access elements of the dataframe: we've shown how to access columns by the `[name of column]` method, and rows via the `.loc[index]` method; and how we can select a range. There are also `.iloc` methods to select by number rather than name; you should become familiar with these on the documentation page for pandas.

3.7 Comparing segments

Earlier, we compared some summary statistics of single people and couples. Let's look at the wider curve for more than one group, now:

```
#This is going to throw a load of errors
income[['Single adult', 'Lone parent, one child under 14']].plot()
```

`TypeError: no numeric data to plot`

3.8 Warning

This isn't looking good. There's a load of text and no graph. If you've not seen this before, it's an error - something has gone wrong. Generally, if we look at the **final** line, it should tell us what's wrong, in this case there's "no numeric data to plot", which is weird, because we've seen the data and have even plotted some of it.

3.9 Messy Data

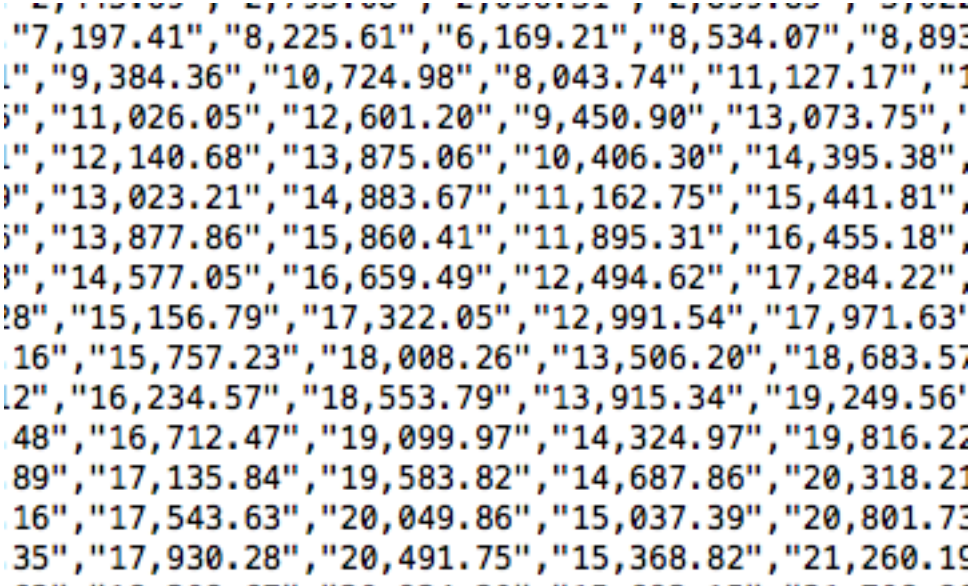
DataFrames, as we are starting to see, give us the chance to plot, chop, slice and data to help us make sense of it. Here, we will create a **new** DataFrame to take only two columns of data, and get rid of any blank cells and any cells which are not being read as numbers - normally a sign of a missing value or a non-numerical character. Why could this be happening? It could be

- due to blank spaces in the text file
- due to letters where there should be numbers
- due to characters (" ", "-", etc) that shouldn't really be there

In general, there will be some detective work required to figure out what's wrong in our text file. Your best bet is sometimes to open up the data in a text editor, like I've done here:

```
from IPython.display import Image

data_path = "https://s3.eu-west-2.amazonaws.com/qm2/wk2/data.png"
Image(data_path)
```



That's a screenshot of our datafile, opened up in a text editor. As we can see, these numbers are separated by commas and surrounded by quotation marks - this is normal, and what .csv files are supposed to look like. However, there are a lot of commas within the numbers - which makes it easier for people to read, but confuses software. Luckily, Python has a method for dealing with this - the “replace” method.

Unfortunately, this dataframe is quite messy, so I'm going to have to extract just the columns of data I'm interested in to make it work. I'll do that by creating a new dataframe:

3.10 Example: Cleaning data

```
clean = income[['Childless couple, annual income','Couple, two children under 14']]
clean.head()
```

Percentile Point	Childless couple, annual income	Couple, two children under 14
1	1,746.92	2,445.69
2	5,141.01	7,197.41
3	6,703.11	9,384.36
4	7,875.75	11,026.05
5	8,671.91	12,140.68

We see those pesky commas. Now we can get on with cleaning up the data:

```
clean=clean.replace(',', '', regex=True)

# In addition, missing values are sometimes written as '-', in order for Python to understand
# value, all '-' need to be replaced with 'NaN'.
clean = clean.replace('-', 'NaN', regex=True).astype('float')
clean.head()
```

Percentile Point	Childless couple, annual income	Couple, two children under 14
1	1746.92	2445.69
2	5141.01	7197.41
3	6703.11	9384.36
4	7875.75	11026.05
5	8671.91	12140.68

Extension: “**Regex**” refers to “**Regular Expression**”, which is a way of replacing and cleaning text. It’s a bit beyond the scope of this class, but worth looking into if you’re interested in programming more widely.

This seems to have done the job. We’ve also put a line in the code to get rid of dashes - a way that data collectors will sometimes represent missing data. Now let’s plot this.

3.11 Asking more questions of the data

For me, this data starts to beg further questions. How would we answer these?

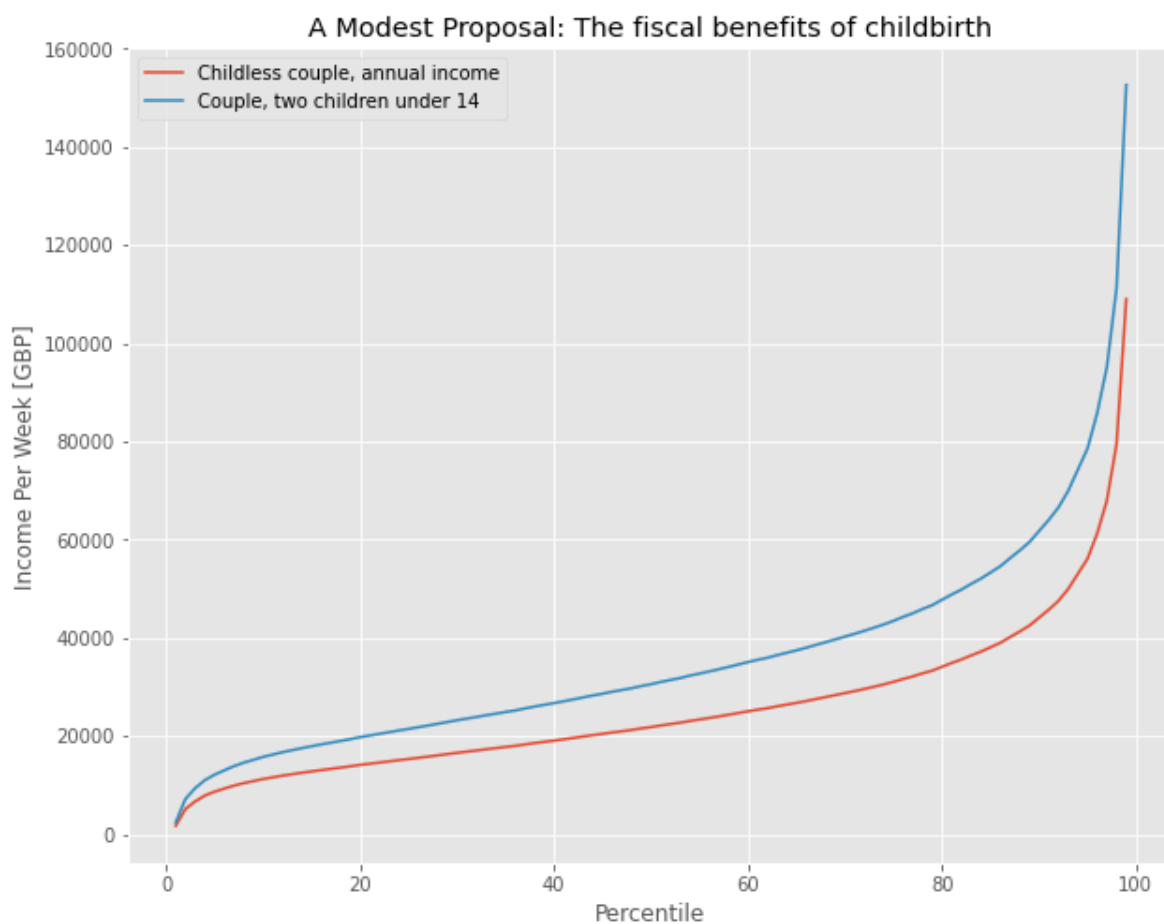
- If the top 20% of income shows such a sharp increase, how do we know that there isn’t a similar uptick *within* the 1%? We’ve already seen that the mean of the dataset as a whole is much less than the half the maximum category (it’s 25% of the maximum).

What if that's true within the 1%, and £2,000/week as a fraction of the 0.1%, or the 0.01%?

- How does this break down for gender, or educational background, or other factors like ethnicity or country of origin?
- Which parts of the income curve show greater gaps between these subgroups and what might it say about the underlying causal mechanisms?

```
clean.plot()
plt.title('A Modest Proposal: The fiscal benefits of childbirth')
plt.xlabel('Percentile')
plt.ylabel('Income Per Week [GBP]')
```

```
Text(0, 0.5, 'Income Per Week [GBP]')
```



3.12 Exercise:

Previously, we'd examined income gaps between single people and couples (how very romantic). Repeat the above exercise (cleaning and plotting income data) for the columns we used above for single people and childless couples. Reflect and comment on the differences.

```
print("Enter your code here")
```

```
Add your reflection here.
```

So far, we've dealt with selecting data in a particular row of column by index or label. What if we now want to filter the data by *value*? For example, let's say I want to see the data for all Childless couples who earn more than 50,000 (net equivalised) pounds every year. This looks like:

```
clean = income[['Childless couple, annual income', 'Couple, two children under 14']]
clean = clean.replace(',', '', regex=True)
clean = clean.replace('-', 'NaN', regex=True).astype('float')
clean[clean['Childless couple, annual income']>50000]
```

The key line of code for selection is:

```
clean[clean['Childless couple, annual income']>50000]
```

Let's break this down: we're used to using `dataframe[some selection]` from earlier. Here "some selection" is

```
clean['Childless couple, annual income']>50000
```

In other words, this command is returning a set of indices where that statement is true. We can see this explicitly:

```
clean['Childless couple, annual income']>50000
```

So python is picking the values where this statement is true - i.e. where the 'Childless couple...' column has values greater than 50000. Then this selection is passed to the dataframe, and the dataframe shows the correct rows.

We won't dwell on comparative operative, here we've used ">" to mean "is greater than"; you can also use:

- == to mean 'is equal to' [why the double equals?]
- <> or != to mean 'is not equal to'

- $<$ to mean ‘is less than’
- the symbol \geq to mean ‘is greater than or equal to’
- \leq to mean ‘is less than or equal to’

3.13 Exercise

On an appropriately labelled graph, plot the incomes of all single adults whose net equivalised income is less than or equal to £10,000. What proportion of the population is this?

4 Extension: Web Scraping

In this example, we've been working with a .csv file that contains all the data we want. That's not always the case. Let's say we're interested in getting the data from a table on a website. Websites are built using HTML code, so what we need to figure out how to look inside the website's code and pull out the data we want. Luckily, pandas has a built in function that can automatically recognize HTML tables in websites and turn them into dataframes.

Let's start with the [Netflix Top 10](https://top10.netflix.com/) website. Click on the link and have a look around. You'll notice two tables: the first showing the top 10 films this week, and the second (farther down) showing the most popular films based on their first 28 days on netflix.

We can download both of these tables into python using one pandas function: `read_html`

```
url='https://top10.netflix.com/'

tables=pandas.read_html(url)

print(tables)
```

```
[  # \
0   1
1   2
2   3
3   4
4   5
5   6
6   7
7   8
8   9
9  10
```

```
.css-ld8rqr-container{position:relative;box-sizing:border-box;min-width:0;}.css-7pg0cj-a11
0           Luckiest Girl Alive
1           Mr. Harrigan's Phone
2           Last Seen Alive
3           Blonde
```

4		Lou
5		The Boss Baby
6		Sing
7		Marauders
8		The Redeem Team
9		Minions & More Volume 1

	Weeks in Top 10	Hours viewed
0	1	43080000
1	1	35420000
2	2	18810000
3	2	17410000
4	3	12600000
5	1	8510000
6	1	8420000
7	2	8350000
8	1	7850000
9	3	7090000 , # \
0	1	
1	2	
2	3	
3	4	
4	5	
5	6	
6	7	
7	8	
8	9	
9	10	

	.css-ld8rqy-container{position:relative;box-sizing:border-box;min-width:0;}.css-7pg0cj-a11
0	Red Notice
1	Don't Look Up
2	Bird Box
3	The Gray Man
4	The Adam Project
5	Extraction
6	Purple Hearts
7	The Unforgivable
8	The Irishman
9	The Kissing Booth 2

	Hours viewed in first 28 days
0	364020000

```

1          359790000
2          282020000
3          253870000
4          233160000
5          231340000
6          228690000
7          214700000
8          214570000
9          209250000 ]

```

When we print the results of what was scraped, it's pretty ugly. One of the reasons is that the `tables` variable is actually a *list* of dataframes. Because there were two tables on our website, `read_html` has returned both of those tables and put them in a list. let's save the first table as a new dataframe called `top10` and have a closer look.

```

top10=tables[0]
top10

```

This looks more like the dataframes we were looking at earlier. There's a big chunk of text (this is HTML code, the language websites are built with) where the name of the second column should be. `read_html` is usually pretty smart, and can actually read the column names from the tables on the website. It seems to have gotten confused for this one column. If we print the columns from the `We` can rename that column using the `rename` function. Since we know it's the second column, we can select it with `top10.columns[1]`

```

top10.rename(columns={top10.columns[1]: "Title" }, inplace = True)
top10

```

And there we have it; a nicely formatted dataframe ready for analysis, straight from a website.

4.1 Exercise

Using the following URL https://en.wikipedia.org/wiki/List_of_Nobel_laureates_in_Chemistry create a plot of the top 10 countries in terms of nobel laureates. First, follow the steps below:

```

# scrape the table of Nobel Laureates in Chemistry using read_html. remember, this gives u

# select the first dataframe from this list and call it chem

```

I'll help you out with this next bit. We'll be using the `groupby` function in pandas to group our dataframe such that each row is a country (rather than a person, as it currently is). We do this by using `<dataframe>.groupby('<column name>')`. Since we're aggregating, we need to tell python how we want it to aggregate our values. In this case, we just want to count the number of rows for each country; we can do this using `.size()`. You can use many different aggregation functions, e.g. `.mean()` if you wanted to calculate the average of a specific column.

```
# here, we're creating a new dataframe called 'country' in which each row is a country, and
countries=chem.groupby('Country[B]').size()

#now let's sort it in descending order
countries.sort_values(ascending=False)

# finally, plot the top 10 countries
```