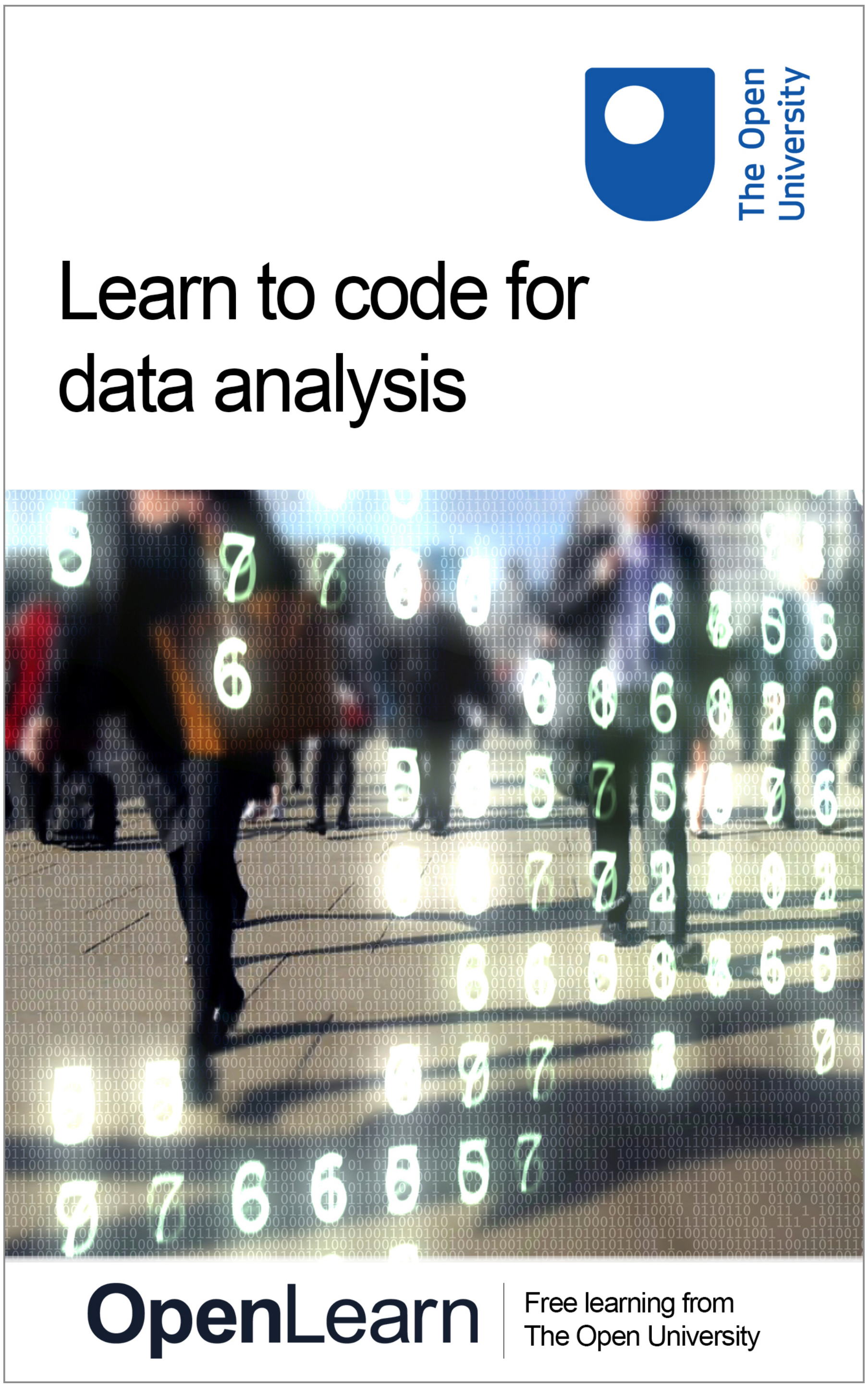
Introduction and guidance



The Open University

Learn to code for data analysis

OpenLearn | Free learning from The Open University

Contents

Week 1: Having a go at it Part 1 7
1 Install the software 7
1.1 Start with a question 8
1.2 Variables and assignments 9
1.3 The art of naming 11
1.4 Downloading the notebook and trying the first exercise 13
1.5 Expressions 14
1.6 Functions 16
1.7 Comments 18
1.8 Values have units 20
2 This week’s quiz 22
3 Summary 23
Week 2: Having a go at it Part 2 25
1 Enter the pandas 25
1.1 This week’s data 27
1.2 Loading the data 28
1.3 Selecting a column 29
1.4 Calculations on a column 31
1.5 Sorting on a column 32
1.6 Calculations over columns 34
2 Writing up the analysis 37
2.1 Practice project 38
2.2 Sharing your project notebook 39
3 This week’s quiz 41
4 Summary 42
4.1 Week 1 and 2 glossary 42
Week 3: Cleaning up our act Part 1 46
Introduction 46
1 Weather data 46
1.1 What is a CSV file? 47
1.2 Dataframes and the ‘dot’ notation 49
1.3 Getting and displaying dataframe rows 50
1.4 Getting and displaying dataframe columns 52
1.5 Comparison operators 55
1.6 Bitwise operators 57
1 Weather data 60
1.1 What is a CSV file? 60
1.2 Dataframes and the ‘dot’ notation 62
1.3 Getting and displaying dataframe rows 63

1.4 Getting and displaying dataframe columns 65
1.5 Comparison operators 68
1.6 Bitwise operators 70
2 This week's quiz 73
3 Summary 74

Week 4: Cleaning up our act Part 2 76
1 Loading the weather data 76
1.1 Removing rogue spaces 77
1.2 Removing extra characters 79
1.3 Missing values 81
1.4 Changing the value types of columns 82
2 Every picture tells a story 88
2.1 Changing a dataframe's index 89
2.2 The project 93
3 This week's quiz 95
4 Summary 96
4.1 Week 4 glossary 97

Week 4: Cleaning up our act Part 2 100
1 Loading the weather data 100
1.1 Removing rogue spaces 101
1.2 Removing extra characters 103
1.3 Missing values 105
1.4 Changing the value types of columns 106
2 Every picture tells a story 112
2.1 Changing a dataframe's index 113
2.2 The project 117
3 This week's quiz 119
4 Summary 120
4.1 Week 4 glossary 121

Week 5: Combine and transform data Part 1 124
1 Life expectancy project 124
1.1 Creating the data 125
1.2 Defining functions 127
1.3 What if...? 129
1.4 Applying functions 131
2 This week's quiz 134
3 Summary 135

Week 6: Combine and transform data Part 2 137
1 Joining left, right and centre 137
1.1 Constant variables 139
1.2 Getting real 141

1.3 Cleaning up 143
1.4 Joining and transforming 146
2 Correlation 150
2.1 Scatterplots 151
2.2 My project 153
3 This week’s quiz 156
4 Summary 157
4.1 Weeks 5 and 6 glossary 157

Week 7: Further techniques Part 1 161

1 I spy with my little eye 161
1.1 Ways of grouping data 163
1.2 Data that describes the world of trade 164
1.3 Exploring the world of export data 165
1.4 Getting data from the Comtrade API 167
1.5 Practice getting data 169
2 This week’s quiz 170
3 Summary 171

Week 8: Further techniques Part 2 173

1 The split-apply-combine pattern 173
1.1 Splitting a dataset by grouping 174
1.2 Looking at apply and combine operations 177
1.3 Summary operations 178
1.4 Filtering groups 181
2 Pivot tables 185
2.1 Pivot tables in pandas 186
2.2 Looking at the milk and cream trade 191
2.3 Your project 192
3 This week’s quiz 194
4 Summary 195
4.1 Week 7 and 8 glossary 195
4.2 What next? 197
Tell us what you think 199
Acknowledgements 199

Week 1: Having a go at it Part 1

1 Install the software

To code in the course notebooks that Ruth mentioned in the video you'll need to install some software.

We're going use a program called Jupyter that opens in your web browser and allows you to write notebooks that include Python code. Jupyter and other software you will need to take part in the course are freely available and you have two options.

Online CoCalc service

The advantages of using CoCalc are that you don't have to install any software and you can work on the course exercises from anywhere there is an internet connection. The disadvantages are that you will need a good internet connection, running the code in your notebook may take time if there are many simultaneous users on CoCalc and you may occasionally lose the latest changes you make in your notebook, because the service will periodically reset.

However, since notebooks are regularly auto-saved, the risk of losing work should be rather small. CoCalc offers a paid plan that has better performance and stability than the free plan. The Open University and the authors have no commercial affiliation with CoCalc.

Install Anaconda package

The other option is to install on your laptop or desktop the free Anaconda package, which includes all necessary software for this course. If you plan to work on this course from multiple computers, you will need to install Anaconda on each one. You can use cloud storage, like Dropbox, to keep your notebooks in sync across machines. Anaconda doesn't have the limitations of CoCalc, so we recommend you use Anaconda if you are going to work on this course always from the same computer.

You should now read the instructions for installing Anaconda or creating a CoCalc project for this course. Don't forget to test everything is working, as explained in the instructions.

The installation of Anaconda is different for Windows, Macs and Linux. Please follow the appropriate instructions.

We advise you to accept the pre-filled defaults suggested during the installation process.

Notebooks

Each week you will use two notebooks (and any necessary data files): an exercise notebook and a project notebook. The notebooks are this course's programming environment, where you will do your own coding.

The exercise notebook contains all the code shown throughout the week, so that you can try it out for yourself, any time you wish. The exercise notebook also contains all the week's exercises. You will be able to solve several exercises just by slightly modifying our code.

The project notebook contains the week's written-up data analysis project, including all necessary code. If you have the extra time, you're encouraged to modify the project notebooks to write up your own data analyses.

You should now download from here the notebooks and data files needed for the whole course.

You'll open the notebooks using Jupyter, which is part of Anaconda and CoCalc. You will learn how to use notebooks later this week, after you've seen what Python code looks like.

Note: please ensure that you abide by any terms and conditions associated with these pieces of software.

1.1 Start with a question

Data analysis often starts with a question or with some data.

Figure 1

A question leads to data that can answer it, and looking at the available data helps to make a question precise or may trigger new questions, which, in turn, may require further data. Data analysis is thus often an iterative process: the questions determine which data to obtain, and the data influences which questions to ask and what the scope of the analysis is. How this week's project came about is an example of such an iterative process.

I (Michel) was watching a news programme mentioning the fight against tuberculosis (TB) as part of the United Nations Millenium Development Goals. Wishing to know how serious

TB is, I browsed the World Health Organization (WHO) website and found a dataset with the number of TB cases and deaths per country per year, from 2007 to 2013. This in turn raised the question of whether a high (or low) number could be mainly due to the country having a large (or small) population. Some more browsing revealed the WHO also has population data from 1990 to 2013.

That was enough data for the fuzzy question: how serious is TB? It was time to make it precise. I chose to measure the effect of TB in terms of deaths, which led to the following questions:

- What is the total, smallest, largest, and average number of deaths due to TB?
- What is the death rate (number of deaths divided by population) of each country?
- Which countries have the smallest and largest number of deaths?
- Which countries have the smallest and largest death rate?

Answering these questions for the whole world and for seven years (2007–2013) would be a bit too much for this initial project. A subset was needed. I decided to take only the latest data for 2013 and, being Portuguese, to focus on the Portuguese-speaking countries. One of them, Brazil, is part of the BRICS group of major emerging economies, so for more diversity the other four countries would be included too: Russia, India, China and South Africa. The project was finally defined! I've added links to the data below if you'd like to take a look!

Activity 1 What would you ask?

Before you embark on coding the analysis to get answers, what other questions could be asked of the datasets described?

What countries would you be interested in? What groups of countries might be interesting to analyse?

Note down some of your questions so that you can come back to them later.

Provide your answer...

WHO POPULATION - DATA BY COUNTRY (LATEST YEAR)
WHO TB MORTALITY AND PREVALENCE - DATA BY COUNTRY (2007 - PRESENT)
Next, I'll explain how I started to organise the information.

1.2 Variables and assignments

With the choice of data and questions confirmed, the coding can begin.

Figure 2

To introduce the basics of coding, I will show you a very simple approach, only suitable for the smallest of datasets. Please bear with me. In the second part of the week I will show you the proper approach. Read through this step and the next – **you're not expected to write code just yet**. In Exercise 1, a bit further on in this week, you'll be asked to start writing code.

Ok, let's start. I want the computer to calculate the total number of deaths in 2013. For the computer to do that, it must first be told what is the number of deaths in each country in that year. I'll start with my home country.

In []:

```
deathsInPortugal = 100
```

The 'In[]' line is Jupyter's way of saying that what follows is code I typed in. And there it is: the first line of code! It is a command to the computer that could be translated to English as: 'find in the attic an empty box, put the number 100 in the box, and write "deathsInPortugal" on the box'. (Aren't you glad Python is more succinct than English?) In coding jargon, the attic is the computer's memory, boxes are called **variables** (I'll explain why shortly), what's written on a box is the variable's **name** , and storing a value in a variable is called an **assignment**.

By naming the boxes, I can later ask the computer to show the value in box **`thingamajig`** or take the values in boxes **`stuff`** and **`moreStuff`** and add them together.

To see what's inside a box, I can just write the name of the box on a line of its own. Jupyter will write the variable's value on the screen, preceded by 'Out[]', to clearly mark the output generated by the code. When you start to use the Jupyter notebooks, you will see numbers inside the square brackets, i.e. In[1], In[2], etc., to indicate in which order the various pieces of code are being executed. Here we have omitted the numbers to avoid confusion between what you see here and what you see in your notebook.

In []:

```
deathsInPortugal
```

Out[]:

```
100
```

Each assignment is written on a line of its own. The computer executes the assignments line by line, from top to bottom. Thus, the program would continue as follows:

In []:

```
deathsInPortugal = 100
deathsInAngola = 200
deathsInBrazil = 300
```

I don't think I need to continue, you get the gist.

By the way, all numbers so far are fictitious. If I use real data, taken from the World Health Organization website, you'll see a difference.

In []:

```
deathsInPortugal = 140
deathsInAngola = 6900
deathsInBrazil = 4400
deathsInPortugal
```

Out[]:

```
140
```

Notice what happened. When a value is assigned to an already existing variable, the value stored in that variable is unceremoniously chucked away and replaced by the new value. In the example, the second group of assignments replaced the values assigned by the first group and thus the current value of **`deathsInPortugal`** is 140 and no longer 100. That's why the storage boxes are called variables: their content can vary over time.

To sum up, a **variable** is a named storage for values and an **assignment** takes the value on the right hand side of the equal sign (=) and stores it in the variable on the left-hand side.

In the next section, you will find out the importance of naming in Python.

1.3 The art of naming

Python is relatively flexible about what you name your variables but rather picky about the format of names.

Figure 3

I could have chosen **`deaths_in_Brazil_in_2013, deathsBrazil,DeathsBrazil, dB`** or even **`stuff`** for my variables. If a box in your attic were labeled **`dB`** or **`stuff`** though, would you know what it contains a year later? So, although you can, it's better not to use cryptic, general, or very long names.

You can't use spaces to separate words in a name, you can't start a name with a digit and names are case-sensitive, i.e. **`deathsBrazil`** and **`DeathsBrazil`** are not the same variable. Making one of those mistakes will result in a **syntax error** (when the computer doesn't understand the line of code) or a **name error** (when the computer doesn't know of any variable with that name).

Let's see some examples. (Remember that you're not expected to write any code for this step.) The first example has spaces between the words, the second example has a digit at the start of the name, and the third example changes the case of a letter, resulting in an unknown name. The kind of error is always at the end of the error message.

In []:

```
deaths In Portugal = 140
File "<ipython-input-7-ded1a063fe45>", line 1
deaths In Portugal = 140
^
SyntaxError: invalid syntax
```

In []:

```
2013deathsInPortugal = 140
File "<ipython-input-8-af085101fcfc>", line 1
2013deathsInPortugal = 140
^
SyntaxError: invalid syntax
```

In []:

```
deathsinPortugal
```

```
———————————————————————————————————

NameError Traceback (most recent call last)

<ipython-input-9-7d3c81b4fb34> in <module ()

—–> 1 deathsinPortugal

NameError: name ‘deathsinPortugal’ is not defined
```

Note that Jupyter doesn’t write any **`Out[]`** because the code is wrong and thus doesn’t generate any output.

In this course, to make names shorter to help fit lines of code on small screens, we’ll use capitalisation instead of underscores to separate the different words of a name, as shown in the code so far. Such practice is called **camel case** independently of the name having **`oneHump`** (‘dromedary case’ just doesn’t sound good, does it?) or **`moreThanTwoHumps`** . The convention in Python is to start variable names with lower case and we’ll stick to it.

In the next section, download the notebook for this week and work through the first exercise – your first line of code!

1.4 Downloading the notebook and trying the first exercise

So far, I’ve done the coding and you’ve read along. Booooring. It’s time to use the Jupyter notebooks and work on the first exercise in the course.

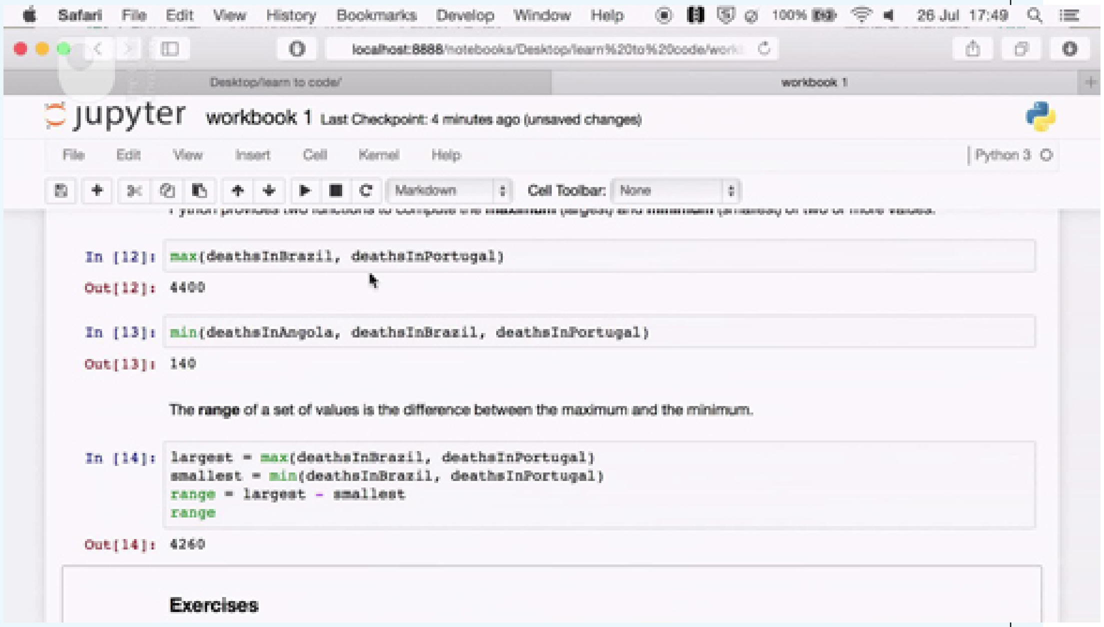
Exercise 1 Variables and assignments

If you haven’t yet installed the software package or created an account on CoCalc, do it now using these instructions!

Open the Exercise notebook 1 (from here), and put it in the folder you created. (You’ll open the data later and learn how to use it in the notebook.)

Once you have installed the file, watch the video to learn how to work with Jupyter notebooks and complete Exercise 1. Pause the video frequently to repeat the demonstrated steps in your notebook. Throughout the week you’ll be directed back to the notebook to complete the other exercises.

Video content is not available in this format.


Safari File Edit View History Bookmarks Develop Window Help
localhost:8888/notebooks/Desktop/learn%20to%20code/work
Desktop/learn to code/
workbook 1
jupyter workbook 1 Last Checkpoint: 4 minutes ago (unsaved changes)
File Edit View Insert Cell Kernel Help
Python 3
Markdown
Cell Toolbar: None

```
In [12]: max(deathsInBrazil, deathsInPortugal)
Out[12]: 4400

In [13]: min(deathsInAngola, deathsInBrazil, deathsInPortugal)
Out[13]: 140
```

The **range** of a set of values is the difference between the maximum and the minimum.

```
In [14]: largest = max(deathsInBrazil, deathsInPortugal)
         smallest = min(deathsInBrazil, deathsInPortugal)
         range = largest - smallest
         range
Out[14]: 4260
```

Exercises

If you haven't yet installed Jupyter and Anaconda, do it now using these instructions.

1.5 Expressions

I've told the computer the deaths in Angola, Brazil and Portugal. I can now ask it to add them together to obtain the total deaths.

In []:

```
deathsInAngola + deathsInBrazil + deathsInPortugal
```

Out[]:

```
11440
```

A fragment of code that has a value is called an **expression**. Calculating the value of an expression is called **evaluating** the expression. If the expression is on a line of its own, the Jupyter notebook displays its value, as above.

The value of an expression can of course be assigned to a variable.

In []:

```
totalDeaths = deathsInAngola + deathsInBrazil + deathsInPortugal
```

Note that no value is displayed because the whole line of code is not an expression, it's a **statement** , a command to the computer. In this case the statement is an assignment. You will see another kind of statement later this week.

To see the value, you learned that you must write the variable's name.

In []:

```
totalDeaths
```

Out[]:

```
11440
```

This is really just a special case of the general rule that writing an expression on its own shows its value. A variable (which stores a value) is just an example of an expression (which is anything that has a value).

I can now write an expression to compute the average number of deaths: it's the total divided by the number of countries.

In []:

```
totalDeaths / 3
```

Out[]:

```
3813.3333333333335
```

Python has of course all four arithmetic **operators** : addition (+), division (/), subtraction (-) and multiplication (*). I'll use the last two later in the week. Python follows the conventional operator precedence: multiplication and division before addition and subtraction, unless parentheses are used to change the order. For example, (3+4)/2 is 3.5 but 3+4/2 is 5.

Figure 4

Now practice writing expressions and complete Exercise 2 in the notebook.

Exercise 2 Expressions

Go back to the Exercise notebook 1 you used in Exercise 1. In Exercise 2 you'll see an example of operator precedence and practise writing expressions.

If you're using Anaconda, remember that to open the notebook you'll need to navigate to it using Jupyter. Whether you're using Anaconda or CoCalc, once the notebook is open, run all the code before doing the exercise.

Writing code for the first time can be difficult but stick with it.

In the next section, you will find out about functions.

1.6 Functions

After the total and the average, next on my to-do list is to calculate the largest number of deaths.

Figure 5

This will be the **maximum**. It takes another single line of code to calculate it.

In []:

```
max(deathsInAngola, deathsInBrazil, deathsInPortugal)
```

Out[]:

```
6900
```

In this expression, `max()` is a function – the parenthesis are a reminder that the name `max` doesn't refer to a variable. A **function** is a piece of code that calculates (**returns**) a value, given zero or more values (the function's **arguments**). In this case, `max()` has three arguments and returns the greatest of them. Actually, `max()` can calculate the maximum of two, three, four or more values.

In []:

```
max(deathsInBrazil, deathsInPortugal)
```

Out[]:

```
4400
```

The expressions above are function **calls**. I'm calling the `max()` function with three or two arguments, and the value of the expression is the value returned by the function. A function is called by writing its name, followed by the arguments, within parentheses and separated by commas. Function names follow the same rules as variable names.

As you might expect, Python also has a function to calculate the smallest (minimum) of two or more values.

In []:

```
min(deathsInAngola, deathsInBrazil, deathsInPortugal)
```

Out[]:

```
140
```

The value returned by a function call can be assigned to a variable. Here is an example, which calculates the **range** of deaths. The range of a set of values is the difference between the largest and the smallest of them.

In []:

```
largest = max(deathsInAngola, deathsInBrazil, deathsInPortugal)
smallest = min(deathsInAngola, deathsInBrazil, deathsInPortugal)
range = largest - smallest
range
```

Out[]:

```
6760
```

Exercise 3 Functions

Identify different types of error (some of you may have experienced those already…) in Exercise 3. You'll need to use the Week 1 notebook to answer question three.

1. If the function name is misspelled as Min, what kind of error is it?

○ A syntax error

Writing `Min(…, …)` instead of `min(…, …)` is not a syntax error, because both have the form of a function call. Names can use uppercase letters.

Take a look at The art of naming.

○ A name error

The computer will understand than `Min(…, …)` is a function call but doesn't know of any function with that name. Remember that names are case-sensitive.

Take a look at The art of naming.

2. If a parenthesis or comma is forgotten, what kind of error is it?

○ A name error

A parenthesis or comma is unrelated to how names are written.

Take a look at The art of naming.

○ A syntax error

A function call requires two parentheses around the arguments, and one comma between successive arguments. Forgetting any of them therefore deviates from the syntax of the Python language.

Take a look at Functions.

---

3. Use Exercise 3 in the Week 1 exercise notebook to answer this question.

What is the range of deaths among the BRICS countries (Brazil, Russia, India, China, South Africa)?

○ 4400

This is the minimum value (for Brazil), not the range.

Take a look at Functions.

○ 65480

This is the average number of deaths, not the range.

Take a look at Expressions.

○ 240000

This is the maximum value (for India), not the range.

Take a look at Functions.

○ 327400

This is the total number of deaths not the range.

Take a look at Functions.

○ 235600

The range is the maximum value (240 thousand for India) minus the minimum value (4400 for Brazil).

1.7 Comments

Last on my to-do list is the death rate, which is the number of deaths divided by the population.

Figure 6

A quick glance at the WHO website tells me Portugal's population in 2013.

In []:

```
populationOfPortugal = 10608
```

Wait a minute! This can't be right. I know Portugal isn't a large country, but ten and a half thousand people is ridiculous. I look more carefully at the WHO website. Oh, the value is given in thousands of people; it's 10 million and 608 thousand people. I could change the assignment to

In []:

```
populationOfPortugal = 10608000
```

but that could give the impression that the population had been counted exactly, whereas it's more likely the number is an estimate based on a previous census. It also makes it easier to check my code against the WHO data if I use the exact same numbers.

I will therefore keep the original assignment but make a note of the unit, using a **comment** , a piece of text that documents what the code does.

In []:

```
# population unit: thousands of inhabitants
populationOfPortugal = 10608
# deaths unit: inhabitants
deathsInPortugal = 140
```

A comment starts with a hash sign **(#)** and goes until the end of the line. Computers ignore all comments, they just execute the code. Comments are your insurance policy: they help you understand your own code if you come back to it after a long break.

I can now compute the death rate, making sure I first convert the population into number of inhabitants, the same unit as deaths.

In []:

```
deathsInPortugal / (populationOfPortugal * 1000)
```

Out[]:

```
1.3197586726998491e-05
```

The death rate (roughly 140 people in 10 million) is a very small number, not very practical to display and reason about. Looking again at the WHO website, I note that other indicators, like TB prevalence, are given per 100 thousand inhabitants. I will do the same for the death rate. Since the population is already in thousands, dividing the deaths by the population gives me the number of deaths per thousand people. Thus, the number of deaths per 100 thousand people must be 100 times higher than that.

In []:

```
# death rate: deaths per 100 thousand inhabitants
deathsInPortugal * 100 / populationOfPortugal
```

Out[]:

```
1.3197586726998491
```

This finishes the basics of coding needed for this week. It took less than 30 lines of code…

Test this out for yourself in Exercise 4 of the Week 1 exercise notebook.

Exercise 4 Comments

Complete the short exercise on the death rate in Exercise 4 in the Week 1 Exercise notebook.

Remember that once the notebook is open, run all the code, before doing the exercise.

1.8 Values have units

Before I move on, let me explain the importance of using comments to record units of measurement.

Figure 7

Values are not just numbers, they have units: degrees Celsius, number of inhabitants, thousands of gallons, etc. Always make a note of the units the value refers to, using comments. This makes it easier to check whether the expressions are right. Disregarding the units will lead to wrong calculations and results.

Activity 2

Have you come across ‘horror stories’ that have happened due to mistakes in the unit of measurement?

Think through what happened and consider what you have learned from them.

Provide your answer...

2 This week’s quiz

Check what you’ve learned this week by taking the end-of-week quiz.

Week 1 practice quiz.

Open the quiz in a new window or tab then come back here when you’ve finished.

3 Summary

The first week of this course covered:

- installing software in course notebooks
- starting data analysis with a question
- the basics of coding
- naming formats
- recording units of measurement.

Next week, you’ll be introduced to pandas. You'll use Jupyter notebooks to write and execute simple programs with Python and the pandas module.

Week 2: Having a go at it Part 2

1 Enter the pandas

As you probably realised, this way of coding is not practical for large scale data analysis.

Figure 1

Three lines of code were required for each country, to store the number of deaths, store the population, and calculate the death rate. With roughly 200 countries in the world, my trivial analysis would require 400 variables and typing almost 600 lines of code! Life's too short to be spent that way.

Instead of using a separate variable for each datum, it is better to organise data as a table of rows and columns.

Table 1

Country	Deaths	Population
Angola	6900	21472

Brazil	4400	200362
Portugal	140	10608

In that way, instead of 400 variables, I only need one that stores the whole table. Instead of writing a mile long expression that adds 200 variables to obtain the total deaths, I'll write a short expression that calculates the total of the 'Deaths' column, no matter how many countries (rows) there are.

To organise data into tables and do calculations on such tables, you and I will use the pandas module, which is included in Anaconda and CoCalc. A **module** is a package of various pieces of code that can be used individually. The pandas module provides very extensive and advanced data analysis capabilities to compliment Python. This course only scratches the surface of pandas.

I have to tell the computer that I'm going to use a module.

In []:

```
from pandas import *
```

That line of code is an **import** statement: from the pandas module, import everything. In plain English: load into memory all pieces of code that are in the pandas module, so that I can use any of them. In the above statement, the asterisk isn't the multiplication operator but instead means 'everything'.

Each weekly project in this course will start with this import statement, because all projects need the pandas module.

The words **`from`** and **`import`** are **reserved words** : they can't be used as variable, function or module names. Otherwise you will get a syntax error.

In []:

```
from = 100

File "<ipython-input-23-6958f0ebc10d>", line 1

from = 100

^

SyntaxError: invalid syntax
```

Jupyter notebooks show reserved words in boldface font to make them easier to spot. If you see a boldface name in an assignment (as you will for the code above), you must choose a different name.

Exercise 5 pandas

Use Exercise 5 the Exercise notebook 1 to help you answer these questions about errors you might come across.

1. What kind of error will you get if you misspell 'pandas' as 'Pandas'?

○ A syntax error

Remember that after the reserved word 'from' comes a module name.

Take a look at The art of naming .

○ A name error, reported as an import error

The computer is expecting a name but there is no module with the name 'Pandas' in the Anaconda distribution. Remember that names are case-sensitive.

2. What kind of error will you get if you misspell 'import' as 'impart'?

○ A name error

A name error only occurs when a name is undefined, but import is not a name, it’s a reserved word.

○ A syntax error

The computer is expecting a reserved word and anything else will raise a syntax error.

3. What kind of error will you get if you forget the asterisk?

○ A name error

An asterisk is not a name so the reported error can’t be this one.

○ A syntax error

The statement cannot end with the reserved word 'import'; the computer is expecting an indication of what to import.

1.1 This week’s data

For the next part of the course you’ll need to download a file of data.

Figure 2

I have created a table with all the data necessary for the project and saved it in an Excel file. Excel is a popular application to create, edit and analyse tabular data. You won’t need Excel to complete this course, but many datasets are provided as Excel files.

Open the data file WHO POP TB some.xls . The file is encoded using UTF-8, a character encoding that allows for accented letters. Do **not** open or edit the file, as you may change how it is encoded, which will lead to errors later on. If you do want to look at its contents, make a copy of the file and look at the copy.

Put the data file in the same folder (or CoCalc project) where you saved your exercise notebook. Done? Great, let's proceed to loading the data – you'll learn how to do this in the next section.

1.2 Loading the data

Many applications can read files in Excel format, and pandas can too. Asking the computer to read the data looks like this:

In []:

```
data = read_excel('WHO POP TB some.xls')
data
```

Out[]:

	Country	Population (1000s)	TB deaths
0	Angola	21472	6900
1	Brazil	200362	4400
2	China	1393337	41000
3	Equatorial Guinea	757	67
4	Guinea-Bissau	1704	1200
5	India	1252140	240000
6	Mozambique	25834	18000
7	Portugal	10608	140
8	Russian Federation	142834	17000
9	Sao Tome and Principe	193	18
10	South Africa	52776	25000
11	Timor-Leste	1133	990

The variable name data is not descriptive, but as there is only one dataset in our analysis, there is no possible confusion with other data, and short names help to keep the lines of code short.

The function `read_excel()` takes a file name as an argument and returns the table contained in the file. In pandas, tables are called **dataframes** . To load the `data`, I simply call the function and store the returned dataframe in a variable.

A file name must be given as a **string** , a piece of text surrounded by quotes. The quote marks tell Python that this isn't a variable, function or module name. Also, the quote marks state that this is a single name, even if it contains spaces, punctuation and other characters besides letters.

Misspelling the file name, or not having the file in the same folder as the notebook containing the code, results in a **file not found** error. In the example below there is an error in the file name.

In []:

```
data = read_excel('WHO POP TB same.xls')
data
___________________________________________

FileNotFoundError Traceback (most recent call last)
<ipython-input-25-c017b2500afa> in <module>()
—> 1 data = read_excel('WHO POP TB same.xls')
2 data
/Users/mw4687/anaconda/lib/python3.4/site-packages/pandas/io/excel.py in read_excel
(io, sheetname, **kwds)
130 engine = kwds.pop('engine', None)
131
-> 132 return ExcelFile(io, engine=engine).parse(sheetname=sheetname, **kwds)
133
134
/Users/mw4687/anaconda/lib/python3.4/site-packages/pandas/io/excel.py in __init__
(self, io, **kwds)
167 self.book = xlrd.open_workbook(file_contents=data)
168 else:
-> 169 self.book = xlrd.open_workbook(io)
170 elif engine == 'xlrd' and isinstance(io, xlrd.Book):
171 self.book = io
/Users/mw4687/anaconda/lib/python3.4/site-packages/xlrd/__init__.py in
open_workbook(filename, logfile,
verbosity, use_mmap, file_contents, encoding_override, formatting_info, on_demand,
ragged_rows)
392 peek = file_contents[:peeksz]
393 else:
-> 394 f = open(filename, "rb")
395 peek = f.read(peeksz)
396 f.close()
FileNotFoundError: [Errno 2] No such file or directory: 'WHO POP TB same.xls'
```

Jupyter notebooks show strings in red. If you see red characters until the end of the line, you have forgotten to type the second quote that marks the end of the string.

In the next section, find out how to select a column.

1.3 Selecting a column

Now you have the data, let the analysis begin!

Figure 3

Let’s tackle the first part of the first question: ‘What are the total, smallest, largest and average number of deaths due to TB?’ Obtaining the total number will be done in two steps: first select the column with the TB deaths, then sum the values in that column.

Selecting a single column of a dataframe is done with an expression in the format: **`dataFrame['column name']`.**

In []:

```
data['TB deaths']
```

Out[]:

```
0 6900
1 4400
2 41000
3 67
4 1200
5 240000
6 18000
7 140
8 17000
9 18
10 25000
11 990
Name: TB deaths, dtype: int64
```

Strings are verbatim text, which means that the column name must be written exactly as given in the dataframe, which you saw after loading the data. The slightest deviation leads to a **key error** , which can be seen as a kind of name error. You can try out in the Week 2 exercise notebook what happens when misspelling the column name. The error message is horribly long. In such cases, just skip to the last line of the error message to see the type of error.

Put this learning into practice in Exercise 6.

Exercise 6 selecting a column

In your Exercise notebook 1, select the population column and store it in a variable, so that you can use it in later exercises.

Remember that to open the notebook you'll need to launch Anaconda and then navigate to the notebook using Jupyter. Once it's open, run all the code.

Next, you'll learn about making calculations on a column.

1.4 Calculations on a column

Having selected the column with the number of deaths per country, I'll add them with the appropriately named sum() method to obtain the overall total deaths.

A **method** is a function that can only be called in a certain context. In this course, the context will mostly be a dataframe or a column. A **method call** looks like a function call, but adds the context in which to call the method: **`context.methodName(argument1, argument2, ...)`** . In other words, a dataframe method can only be called on dataframes, a column method only on columns. Because methods are functions, a method call returns a value and is therefore an expression.

If all that sounded too abstract, here's how to call the **`sum()`** method on the TB deaths column. Note that **`sum()`** doesn't need any arguments because all the values are in the column.

In []:

```
tbColumn = data['TB deaths']
tbColumn.sum()
```

Out[]:

```
354715
```

The estimated total number of deaths due to TB in 2013 in the BRICS and Portuguese-speaking countries was over 350 thousand. An impressive number, for the wrong reasons.

Calculating the minimum and maximum number of deaths is done in a similar way.

In []:

```
tbColumn.min()
```

Out[]:

```
18
```

In []:

```
tbColumn.max()
```

Out[]:

```
240000
```

Like **`sum()`** , the column methods **`min()`** and **`max()`** don't need arguments, whereas the Python functions **`min()`** and **`max()`** did need them, because there was no context (column) providing the values.

The average number is computed as before, dividing the total by the number of countries.

In []:

```
tbColumn.sum() / 12
```

Out[]:

```
29559.583333333332
```

This kind of average is called the **mean** and there's a method for that.

In []:

```
tbColumn.mean()
```

Out[]:

```
29559.583333333332
```

Another kind of average measure is the **median** , which is the number in the middle, i.e. half of the values are above the median and half below it.

In []:

```
tbColumn.median()
```

Out[]:

```
5650.0
```

The mean is five times higher than the median. While half the countries had less than 5650 deaths in 2013, some countries had far more, which pushes the mean up.

The median is probably closer to the intuition you have of what 'average' should mean (pun intended). News reports don't always make clear what average measure is being used, and using the mean may distort reality. For example, the mean household income in a country will be influenced by very poor and very rich households, whereas the median income doesn't take into account how poor or rich the extremes are: it will always be half the households below and half above the median.

Put this learning into practice in Exercise 7.

Exercise 7 calculations on a column

Practise the use of column methods by applying them to the population column you obtained in Exercise 6 in the Exercise notebook 1. Remember to run all code before doing the exercise.

1.5 Sorting on a column

One of the research questions was: which countries have the smallest and largest number of deaths?

Being a small table, it is not too difficult to scan the TB deaths column and find those countries. However, such a process is prone to errors and impractical for large tables. It's much better to sort the table by that column, and then look up the countries in the first and last rows.

As you've guessed by now, sorting a table is another single line of code.

In []:

```
data.sort_values('TB deaths')
```

Out[]:

	Country	Population (1000s)	TB deaths
9	Sao Tome and Principe	193	18
3	Equatorial Guinea	757	67
7	Portugal	10608	140
11	Timor-Leste	1133	990
4	Guinea-Bissau	1704	1200
1	Brazil	200362	4400
0	Angola	21472	6900
8	Russian Federation	142834	17000
6	Mozambique	25834	18000
10	South Africa	52776	25000
2	China	1393337	41000
5	India	1252140	240000

The dataframe method **`sort_values()`** takes as argument a column name and returns a new dataframe where the rows are in ascending order of the values in that column. Note that sorting doesn’t modify the original dataframe.

In []:

```
data # rows still in original order
```

Out[]:

	Country	Population (1000s)	TB deaths
0	Angola	21472	6900
1	Brazil	200362	4400
2	China	1393337	41000
3	Equatorial Guinea	757	67
4	Guinea-Bissau	1704	1200
5	India	1252140	240000
6	Mozambique	25834	18000
7	Portugal	10608	140
8	Russian Federation	142834	17000

9	Sao Tome and Principe	193	18
10	South Africa	52776	25000
11	Timor-Leste	1133	990

It's also possible to sort on a column that has text instead of numbers; the rows will be sorted in alphabetical order.

In []:

```
data.sort_values('Country')
```

Out[]:

	Country	Population (1000s)	TB deaths
0	Angola	21472	6900
1	Brazil	200362	4400
2	China	1393337	41000
3	Equatorial Guinea	757	67
4	Guinea-Bissau	1704	1200
5	India	1252140	240000
6	Mozambique	25834	18000
7	Portugal	10608	140
8	Russian Federation	142834	17000
9	Sao Tome and Principe	193	18
10	South Africa	52776	25000
11	Timor-Leste	1133	990

Exercise 8 sorting on a column

Use the Exercise notebook 1 to sort the table by population so that you can quickly see which are the least and the most populous countries. Remember to run all code before doing the exercise.

In the next section you'll learn about calculations over columns.

1.6 Calculations over columns

The last remaining task is to calculate the death rate of each country.

You may recall that with the simple approach I'd have to write:

```
rateAngola = deathsInAngola * 100 / populationOfAngola
```

```
rateBrazil = deathsInBrazil * 100 / populationOfBrazil
```

and so on, and so on. If you’ve used spreadsheets, it’s the same process: create the formula for the first row and then copy it down for all the rows. This is laborious and error-prone, e.g. if rows are added later on. Given that data is organised by columns, wouldn’t it be nice to simply write the following?

`rateColumn = deathsColumn * 100 / populationColumn`

Say no more: your wish is pandas’s command.

In []:

```
deathsColumn = data['TB deaths']
populationColumn = data['Population (1000s)']
rateColumn = deathsColumn * 100 / populationColumn
rateColumn
```

Out[]:

```
0 32.134873
1 2.196025
2 2.942576
3 8.850727
4 70.422535
5 19.167186
6 69.675621
7 1.319759
8 11.901928
9 9.326425
10 47.370017
11 87.378641
dtype: float64
```

Tadaaa! With pandas, the arithmetic operators become much smarter. When adding, subtracting, multiplying or dividing columns, the computer understands that the operation is to be done row by row and creates a new column.

All well and nice, but how to put that new column into the dataframe, in order to have everything in a single table? In an assignment **`variable = expression`** , if the variable hasn’t been mentioned before, the computer creates the variable and stores in it the expression’s value. Likewise, if I assign to a column that doesn’t exist in the dataframe, the computer will create it.

In []:

```
data['TB deaths (per 100,000)'] = rateColumn
data
```

Out[]:

	Country	Population (1000s)	TB deaths	TB deaths (per 100,000)
0	Angola	21472	6900	32.134873
1	Brazil	200362	4400	2.196025

2	China	1393337	41000	2.942576
3	Equatorial Guinea	757	67	8.850727
4	Guinea-Bissau	1704	1200	70.422535
5	India	1252140	240000	19.167186
6	Mozambique	25834	18000	69.675621
7	Portugal	10608	140	1.319759
8	Russian Federation	142834	17000	11.901928
9	Sao Tome and Principe	193	18	9.326425
10	South Africa	52776	25000	47.370017
11	Timor-Leste	1133	990	87.378641

That’s it! I’ve written all the code needed to answer the questions I had. Next I’ll write up the analysis into a succinct and stand-alone notebook that can be shared with friends, family and colleagues or the whole world. You’ll find that in the next section.

2 Writing up the analysis

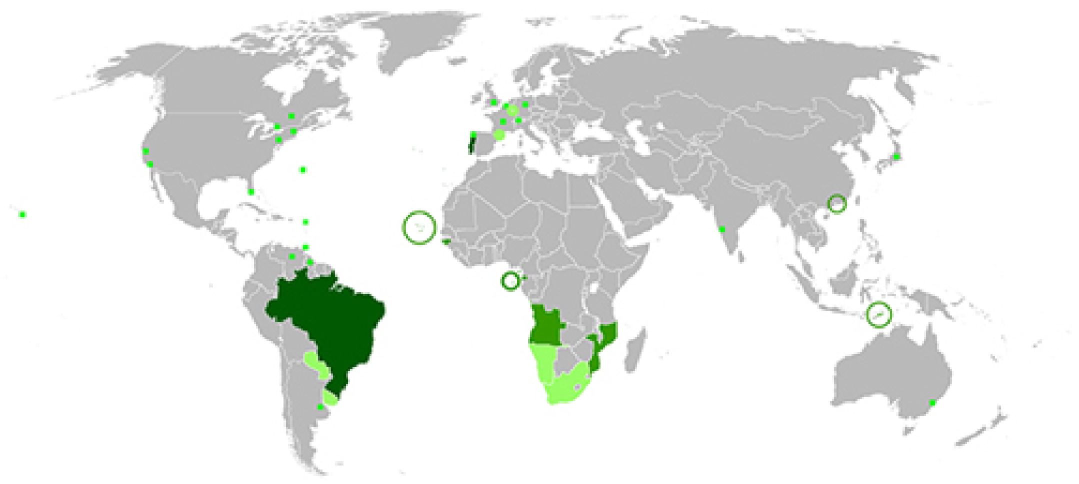


Figure 4 A map identifying Portuguese speaking countries

Once you've done your analysis, you may want to record it or share it with others. The best way is to write up what you've discovered.

There is no right or wrong way to write up data analysis but the important thing is to present the answers to the questions you had. To keep things simple, I suggest the following structure:

1. A descriptive title
2. An introduction setting the context and stating what you want to find out with the data.
3. A section detailing the source(s) of the data, with the code to load it into the notebook.
4. One or more sections showing the processes (calculating statistics, sorting the data, etc.) necessary to address the questions.
5. A conclusion summarising your findings, with qualitative analysis of the quantitative results and critical reflection on any shortcomings in the data or analysis process.

You don't need to explain your code, but it's helpful to write the text in such a way that even readers who know nothing about Python or pandas can follow your analysis.

You can see how I've written up the analysis by opening this week's project notebook, which you can open in project_1: Deaths by tuberculosis .

In the next section, amend this project to produce your own version.

2.1 Practice project

Figure 5

Here's a quick project for you, which is about looking at TB deaths in *all* countries.

Activity 1 The project

1. Open the data file WHO POP TB all.xls . Do **not** open or edit this file, to avoid changing its encoding. If you want to see the contents of the file, make a copy and look at the copy.
2. Open the project notebook.
3. If you're using CoCalc do the following two steps:
   a. Click on the File menu and select 'Download as' and then 'IPython notebook (.ipynb)'.
   b. On your computer, rename the downloaded file so that it includes your name, e.g. 'TB deaths all world – Michel Wermelinger.ipynb'. Then upload the renamed notebook to CoCalc and open it.
4. If you're using Anaconda do the following two steps:
   a. Click on the File menu and select 'Make a copy'.
   b. Click on the title of the new notebook ('project 1-Copy1') to rename it. Make sure to include your name in the file name, e.g. 'TB deaths all world – Michel Wermelinger'.
5. In the new notebook, add your name to mine and update the date.
6. Edit the first code cell: change the file name to 'WHO POP TB all.xls', in order to load the data for all countries in the world.
7. Run all cells in the notebook. This might take a little while.

8. Add one line of code at the end to sort the table by the death rate, so that it's easy to see the least and most affected countries.
9. Go through the notebook and change any text (in particular the conclusions) to reflect the new results.
10. Save and then close and halt the notebook.

If you happen to know how to use a spreadsheet application, then you can do a personal project: open the Excel file, remove all countries you are not interested in, and then do the analysis only for the remaining subset.

You might like to share your experience of working on this project with friends, family or colleagues.

2.2 Sharing your project notebook

Figure 6

Sharing work is a great way to solve problems and learn from others.

You are encouraged to share the analysis notebook that you created in the previous section. There are a few different ways you can do this. I will only mention two, sharing and publishing, depending on whether you want people to be able to change your notebook or only read it.

If you don't mind people editing and extending your notebook, like you have done with mine, then you'll need to give them the notebook file (e.g. 'TB deaths all world – Michel Wermelinger.ipynb') and all necessary data files (just the 'WHO POP TB all.xls' in this case). There are many ways you can share files with other people. One of the simplest is to create a zip archive, upload it to a cloud service like Dropbox or Google Drive, and publicise the download link. You could also share the link on your social media or via email.

If the intended recipients don’t have the necessary software (Python, pandas and Jupyter) or you don’t want anybody to change your notebook, you can still publish the analysis in read-only mode, i.e. people can read the text and code, see the resulting tables and numbers, but can’t modify anything.

To do this, open your project notebook, run all the cells, double-check that there are no error messages and that all values and tables are shown as you want them to be, and save the notebook (without closing it).

If you use Anaconda, export the notebook by clicking ‘Download as’ in the ‘File’ menu and selecting the option you prefer. I prefer HTML because it looks much nicer. You can then share the single PDF or HTML file as before, by email, via Dropbox or Google Drive, on your blog and via a link.

If you use CoCalc, just click on the ‘Publish’ button on the right side above your notebook, and you will get after a little while the link that you can share with others. Anyone can then read your notebook, even if they don’t have a CoCalc account. For example, look at my Project 1 (it’s best to right-click and open this link in a new tab).

Now choose the sharing or publishing method, and get sharing!

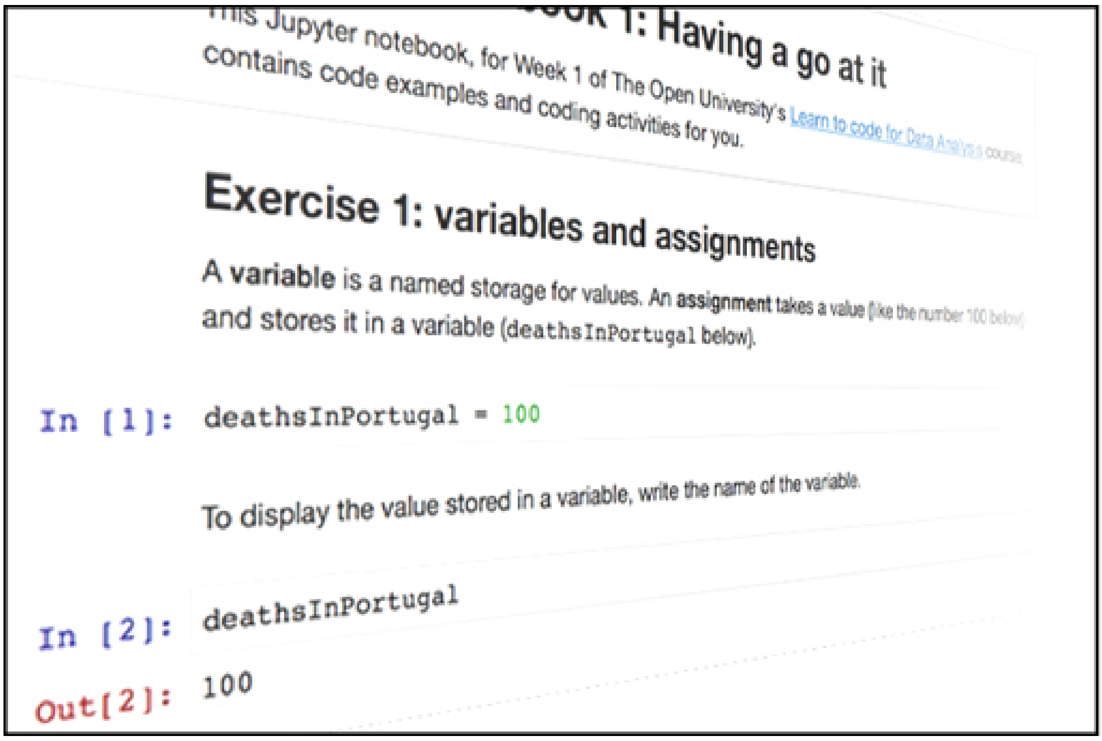
3 This week’s quiz

Check what you’ve learned this week by taking the end-of-week quiz.

Week 2 practice quiz

Open the quiz in a new window or tab then come back here when you’ve finished.

4 Summary


ook 1: Having a go at it

This Jupyter notebook, for Week 1 of The Open University's Learn to code for Data Analysis course, contains code examples and coding activities for you.

Exercise 1: variables and assignments

A **variable** is a named storage for values. An **assignment** takes a value (like the number 100 below) and stores it in a variable (`deathsInPortugal` below).

```
In [1]: deathsInPortugal = 100
```

To display the value stored in a variable, write the name of the variable.

```
In [2]: deathsInPortugal
Out[2]: 100
```

Figure 7

This week you used Jupyter notebooks to write and execute simple programs with Python and the pandas module. You've learned how to:

- load a table from an Excel file
- select a column, and compute some simple statistics (like the total, minimum and median) about it.
- create a new column with values calculated from other columns
- sort a table by one of its columns.

Next week you will learn further ways to manipulate dataframes, in particular to clean data. You will also produce your first data chart, showing variations of values over time.

Futher reading

WHO population – data by country (2013)

WHO mortality and prevalence – data by country (2007 – present)

4.1 Week 1 and 2 glossary

Here are alphabetical lists, for quick look up, of what this week introduced.

Programming and data analysis concepts

An **assignment** is a statement of the form **`variable = expression`** . It evaluates the expression and stores its value in the variable. The variable is created if it doesn't exist. Each assignment is written on its own line.

CamelCase is a naming style in which names made of various words have each word capitalized, except possibly the first.

A **comment** is a note about the code. It starts with the hash sign (#) and goes until the end of the line.

A **dataframe** is the pandas representation of a table.

An **expression** is a fragment of code that can be **evaluated** , i.e. that has a value, like a variable name.

A **file not found** error occurs if the computer can't find the given file, e.g. because the name is misspelled or because it's in another folder.

A **function** takes zero or more **arguments** (values) and **returns** (produces) a value.

A **function call** is an expression of the form `functionName(argument1, argument2, …).`

An **import statement** of the form `from module import` * loads all the code from the given module.

The **maximum** and **minimum** of a set of values is the largest and smallest value, respectively.

The **mean** of a set of numbers is the sum of those numbers divided by how many there are.

The **median** of a set of numbers is the number in the middle, i.e. half of the numbers are below the median and half are above.

A **method** is a function that can only be called in a certain context, like a dataframe or a column.

A **method call** is an expression of the form `context.methodName(argument1, argument2, ...).`

A **module** is a package of various pieces of code that can be used individually.

A **name** is a case-sensitive sequence of letters, digits and underscores. Names cannot start with a digit. Function, variable and module names usually start with lowercase.

A **name error** occurs if the computer doesn't recognize a name, e.g. if it was misspelled.

An **operator** is a symbol that represents some operation on one or two expressions, e.g. the four basic arithmetic operators.

The **range** of a set of values is the difference between the maximum and the minimum.

A **reserved** word cannot be used as a name. Jupyter shows reserved words in green boldface.

A **statement** is a command for the computer to do something, e.g. to assign a value or to import some code.

A **string** is a verbatim piece of text, surrounded by quotes. Jupyter shows strings in red.

A **syntax error** occurs if the computer can't understand the code because it is not in the expected form, e.g. if a reserved word is used instead of a name or some punctuation is missing.

A **variable** is a named storage for values.

Reserved words

- `from`
- `import`

Functions and methods

`max(value1, value2, …)` returns the maximum of the given values.

`column.max()` returns the maximum value in the column.

`min(value1, value2, …)` returns the minimum of the given values.

`column.min()` returns the minimum value in the column.

`column.mean()` returns the mean of the values in the column.

`column.median()` returns the median of the values in the column.

`column.sum()` returns the total of the values in the column.

`dataFrame.sort_values(columnName)` takes a string with a column's name and returns a new dataframe, in which rows are sorted in ascending order according to the values in the given column.

`read_excel(fileName)` takes a string with an Excel file name, reads the file, and returns a dataframe representing the table in the file.

Week 3: Cleaning up our act Part 1

Introduction

Welcome to Week 3.

Please note: in the following video, where reference is made to a study 'week', this corresponds to Weeks 3 and 4 of this course.

Video content is not available in this format.

In Week 1 and 2 you worked on a dataset that combined two different World Health Organization datasets: population and the number of deaths due to tuberculosis.

They could be combined because they share a common attribute: the countries. This week you will learn the techniques behind the creation of such a combined dataset.

1 Weather data

This week you will be looking at investigating historic weather data.

Figure 1

Of course, such data is hugely important for research into the large-scale, long-term shift in our planet's weather patterns and average temperatures – climate change. However, such data is also incredibly useful for more mundane planning purposes. To demonstrate the learning this week, I, Rob Griffiths, will be using historic weather data to try and plan a summer holiday in the UK. You'll use the data too and get a chance to work on your own project at the end of the week.

The dataset we'll use to do this will come from the Weather Underground, which creates weather forecasts from data sent to them by a worldwide network of over 100,000 weather enthusiasts who have personal weather stations on their house or in their garden.

In addition to creating weather forecasts from that data, the Weather Underground also keeps that data as historic weather records allowing members of the public to download weather datasets for a particular time period and location. These datasets are downloaded as CSV files, explained in the next step.

Datasets are rarely 'clean' and fit for purpose, so it will be necessary to clean up the data and 'mould it' for your purposes. You will then learn how to visualise data by creating graphs using the **`plot()`** function.

1.1 What is a CSV file?

A CSV file is a plain text file that is used to hold tabular data. The acronym CSV is short for 'comma-separated values'.

Figure 2

Take a look at the first few lines of a CSV file that holds the same data as the Excel file ‘WHO POP TB all.xls’ that you encountered in Week 2:

```
Country,Population (1000s),TB deaths
Afghanistan,30552,13000.0
Albania,3173,20.0
Algeria,39208,5100.0
Andorra,79,0.26
Angola,21472,6900.0
Antigua and Barbuda,90,1.2
Argentina,41446,570.0
Armenia,2977,170.0
```

Notice that the first line is a row of column names. The subsequent lines are rows of actual data that correspond to the column names. The row of column names is optional, but it is helpful in understanding the data in the following lines and making sure the right values fall in the right place. In this example, the first value on every row must be a string representing a country’s name, the second value is an integer representing that country’s population (in 1000s) and the third value is a decimal representing the number of deaths due to TB. Note that the third value is a decimal (like 0.26 deaths for Andorra) and not an integer because it is an estimate obtained from statistical processing of collected data.

Note that each value or column name is separated by a comma but actually any character can be used to separate values in a CSV file, including spaces and tabs etc., hence CSV can also stand for ‘character-separated values’.

Because CSV files are in plain-text it makes the data easy to import into any spreadsheet program, database or pandas dataframe.

Before anything can be done with a CSV file with pandas, the following import statement must be executed:

In []:

```
from pandas import *
```

As you learned in Week 2, the import statement loads into memory all the code in the pandas module.

To read a CSV file into a dataframe, the pandas function **`read_csv()`** needs to be called.

In []:

```
df = read_csv('WHO POP TB all.csv')
```

The above code creates a dataframe from the data in the file **`WHO POP TB all.csv`** and assigns it to the variable **`df`**. This is the simplest usage of the **`read_csv()`** function, just using a single argument, a string that holds the name of the CSV file.

However the function can take many additional arguments (some of which you'll use later), which determine how the file is to be read.

In the next step, find out about dataframes and the 'dot' notation.

1.2 Dataframes and the 'dot' notation

In Week 2 you learned that dataframes have methods, which are like functions, that can only be called in the context of a dataframe.

For example, because the TB deaths dataframe **`df`** has a column named 'Country', the **`sort_values()`** method can be called like this:

In []:

```
df.sort_values('Country')
```

Because there is variable name, followed by a dot, followed by the method, this is called **dot notation**. Methods are said to be a property of a dataframe. In addition to methods, dataframes have another property – attributes.

Figure 3

Attributes

A dataframe attribute is like a variable that can only be accessed in the context of a dataframe. One such attribute is **`columns`** which holds a dataframe's column names.

So the expression **`df.columns`** evaluates to the value of the **`columns`** attribute inside the dataframe **`df`**. The following code will get and display the names of the columns in the dataframe **`df`**:

In []:

```
df.columns
```

Out[]:

```
Index(['Country', 'Population (1000s)', 'TB deaths'],
dtype='object')
```

1.3 Getting and displaying dataframe rows

Dataframes can have hundreds or thousands of rows, so it is not practical to display a whole dataframe.

However, there are a number of dataframe attributes and methods that allow you to get and display either a single row or a number of rows at a time. Three of the most useful methods are: **`iloc()`**, **`head()`** and **`tail()`**. Note that to distinguish methods and attributes, we write `()` after a method's name.

Figure 4

The iloc attribute

A dataframe has a default integer index for its rows, which starts at 0 (zero). You can get and display any single row in a dataframe by using the**`iloc`** attribute with the index of the row you want to access as its argument. For example, the following code will get and display the first row of data in the dataframe **df**, which is at index 0:

In []:

```
df.iloc[0]
```

Out[]:

```
Country Afghanistan
Population (1000s) 30552
TB deaths 13000
Name: 0, dtype: object
```

Similarly, the following code will get and display the third row of data in the dataframe **df**, which is at index 2:

In []:

```
df.iloc[2]
```

Out[]:

```
Country Algeria
Population (1000s) 39208
TB deaths 5100.0
Name: 0, dtype: object
```

The head() method

The first few rows of a dataframe can be printed out with the **head()** method.

You can tell **head()** is a method, rather than an attribute such as **columns**, because of the parentheses (round brackets) after the property name.

If you don't give any argument, i.e. don't put any number within those parentheses, the default behaviour is to return the first five rows of the dataframe. If you give an argument, it will print that number of rows (starting from the row indexed by 0).

For example, executing the following code will get and display the first five rows in the dataframe **df**.

In []:

```
df.head()
```

Out[]:

	Country	Population (1000s)	TB deaths
0	Afghanistan	30552	13000.00
1	Albania	3173	20.00
2	Algeria	39208	5100.00
3	Andorra	79	0.26
4	Angola	21472	6900.00

And, executing the following code will get and display the first seven rows in the dataframe **df.**

In []:

```
df.head(7)
```

Out[]:

	Country	Population (1000s)	TB deaths
0	Afghanistan	30552	13000.00
1	Albania	3173	20.00
2	Algeria	39208	5100.00
3	Andorra	79	0.26
4	Angola	21472	6900.00
5	Antigua and Barbuda	90	1.20
6	Argentina	41446	570.00

The tail() method

The **`tail()`** method is similar to the **`head()`** method.

If no argument is given, the last five rows of the dataframe are returned, otherwise the number of rows returned is dependent on the argument, just like for the **`head()`** method.

In []:

```
df.tail()
```

Out[]:

	Country	Population (1000s)	TB deaths
189	Venezuela (Bolivarian Republic of)	30405	480
190	Viet Nam	91680	17000
191	Yemen	24407	990
192	Zambia	14539	3600
193	Zimbabwe	14150	5700

1.4 Getting and displaying dataframe columns

You learned in Week 2 that you can get and display a single column of a dataframe by putting the name of the column (in quotes) within square brackets immediately after the dataframe’s name.

For example, like this:

In []:

```
df['TB deaths']
```

You then get output like this:

Out[]:

```
0 13000.00
1 20.00
2  5100.00
3 0.26
4  6900.00
5 1.20
6 570.00
...
```

Notice that although there is an index, there is no column heading. This is because what is returned is not a new dataframe with a single column but an example of the **`Series`** data type.

Figure 5

Each column in a dataframe is an example of a series

The **`Series`** data type is a collection of values with an integer index that starts from zero. In addition, the **`Series`** data type has many of the same methods and attributes as the **`DataFrame`** data type, so you can still execute code like:

In []:

```
df['TB deaths'].head()
```

Out[]:

```
0 13000.00
1 20.00
```

```
2  5100.00
3 0.26
4  6900.00
Name: TB deaths, dtype: float64
```

And

In []:

```
df['TB deaths'].iloc[2]
```

Out[]:

```
5100.00
```

However, pandas does provide a mechanism for you to get and display one or more selected columns as a new dataframe in its own right. To do this you need to use a **list**. A list in Python consists of one or more items separated by commas and enclosed within square brackets, for example **`['Country']`** or **`['Country', 'Population (1000s)']`**. This list is then put within outer square brackets immediately after the dataframe's name, like this:

In []:

```
df[['Country']].head()
```

Out[]:

	Country
0	Afghanistan
1	Albania
2	Algeria
3	Andorra
4	Angola

Note that the column is now named. The expression **`df[['Country']]`**(with two square brackets) evaluates to a new dataframe (which happens to have a single column) rather than a series.

To get a new dataframe with multiple columns you just need to put more column names in the list, like this:

In []:

```
df[['Country', 'Population (1000s)']].head()
```

Out[]:

	Country	Population (1000s)
0	Afghanistan	30552
1	Albania	3173
2	Algeria	39208

3	Andorra	79
4	Angola	21472

The code has returned a new dataframe with just the `'Country'` and `'Population (1000s)'` columns.

Exercise 1 Dataframes and CSV files

Now that you've learned about CSV files and more about pandas you are ready to complete Exercise 1 in the exercise notebook 2.

Open the exercise 2 notebook and the data file you used last week WHO POP TB all.csv and save it in the folder you created in Week 1.

If you're using Anaconda instead of CoCalc, remember that to open the notebook you'll need to navigate to the notebook using Jupyter. Once it's open, run the existing code in the notebook before you start the exercise. When you've completed the exercise, save the notebook. If you need a quick reminder of how to use Jupyter watch again the video in Week 1 Exercise 1.

1.5 Comparison operators

In Expressions,you learned that Python has arithmetic operators: +, /, - and * and that expressions such as 5 + 2 evaluate to a value (in this case the number 7).

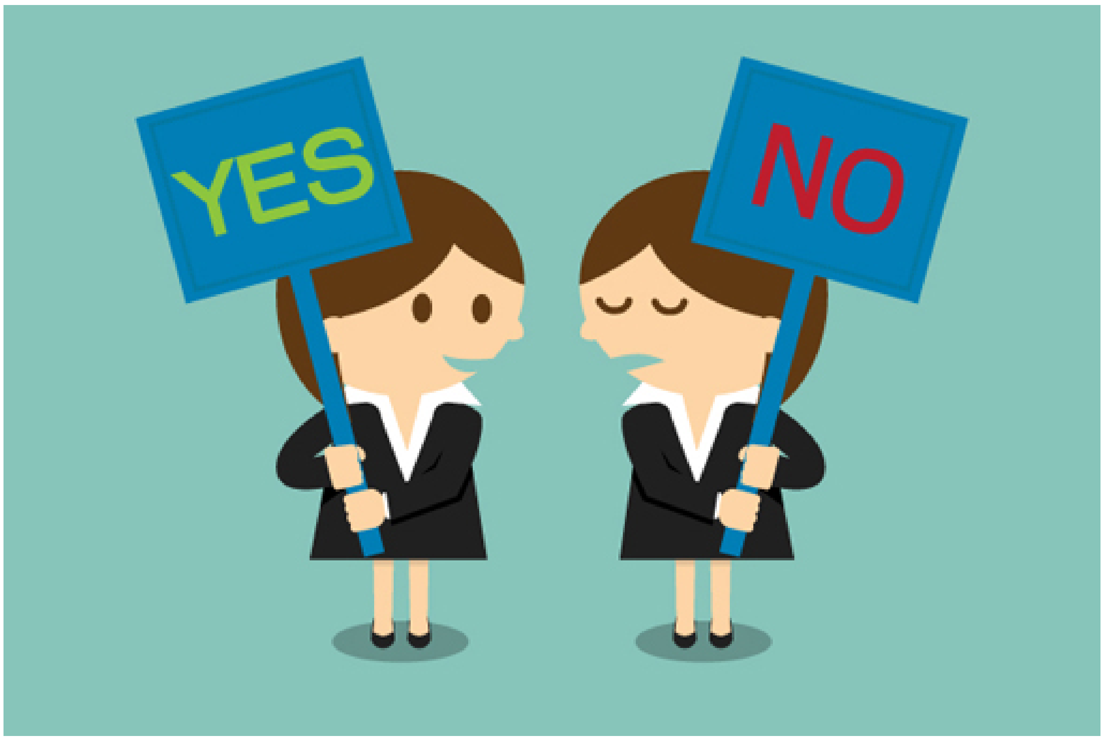


Figure 6

Python also has what are called comparison operators, these are:

```
== equals
!= not equal
< less than
> greater than
<= less than or equal to
>= greater than or equal to
```

Expressions involving these operators always evaluate to a Boolean value, that is **`True`** or **`False`**. Here are some examples:

```
2 = = 2 evaluates to True
2 + 2 = = 5 evaluates to False
2 != 1 + 1 evaluates to False
45 < 50 evaluates to True
20 > 30 evaluates to False
100 <= 100 evaluates to True
101 >= 100 evaluates to True
```

The comparison operators can be used with other types of data, not just numbers. Used with strings they compare using alphabetical order. For example:

`'aardvark' < 'zebra' evaluates to True`

In Calculating over columns you saw that when applied to whole columns, the arithmetic operators did the calculations row by row. Similarly, an expression like **`df['Country'] >= 'K'`** will compare the country names, row by row, against the string 'K' and record whether the result is **`True`** or **`False`** in a series like this:

```
0 False
1 False
2 False
3 False
4 False
5 False
...
Name: Country, dtype: bool
```

If such an expression is put within square brackets immediately after a dataframe's name, a new dataframe is obtained with only those rows where the result is **`True`**. So:

`df[df['Country'] >= 'K']`

returns a new dataframe with all the columns of **`df`** but with only the rows corresponding to countries starting with K or a letter later in the alphabet.

As another example, to see the data for countries with over 80 million inhabitants, the following code will return and display a new dataframe with all the columns of **`df`** but with only the rows where it is **`True`** that the value in the **`'Population (1000s)'`** column is greater than **`80000:`**

In []:

```
df[df['Population (1000s)'] > 80000]
```

Out[]:

	Country	Population (1000s)	TB deaths
13	Bangladesh	156595	80000
23	Brazil	200362	4400
36	China	1393337	41000
53	Egypt	82056	550
58	Ethiopia	94101	30000
65	Germany	82727	300
77	India	1252140	240000
78	Indonesia	249866	64000
85	Japan	127144	2100
109	Mexico	122332	2200
124	Nigeria	173615	160000
128	Pakistan	182143	49000
134	Philippines	98394	27000
141	Russian Federation	142834	17000
185	United States of America	320051	490
190	Viet Nam	91680	17000

Exercise 2 Comparison operators

You are ready to complete Exercise 2 in the Exercise notebook 2.

Remember to run the existing code in the notebook before you start the exercise. When you've completed the exercise, save the notebook.

1.6 Bitwise operators

To build more complicated expressions involving column comparisons, there are two bitwise operators.

Figure 7

The `&` operator means ‘and’ and the | operator (vertical bar, not uppercase letter ‘i’) means ‘or’. So, for example the expression:

```
(df['Country'] >= 'Latvia') & (df['Country'] <= 'Sweden')
```

will evaluate to a series containing Boolean values where the values are**`True`** only if the equivalent rows in the dataframe contain the countries ‘**`Latvia`**’ to ‘**`Sweden`**’, inclusive. However, the following expression which uses | (or) rather than & (and):

```
(df['Country'] >= 'Latvia') | (df['Country'] <= 'Sweden')
```

will evaluate to **`True`** for all countries, because every country comes alphabetically after ‘**`Latvia`**’ (e.g. the ‘UK’) or before '**`Sweden`**' (e.g. ‘**`Brazil`**’).

Note the round brackets around each comparison. Without them you will get an error.

The whole expression with multiple comparisons has to be put within **`df[…]`** to get a dataframe with only those rows that match the condition.

As a further example, using different columns, it is relatively easy to find the rows in **`df`** where '**`Population (1000s)`**' is greater than **`80000`** and where '**`TB deaths`**' are greater than `10000`.

`In []:`

```
df[(df['Population (1000s)'] > 80000) & (df['TB deaths'] > 10000)]
```

`Out []:`

	Country	Population (1000s)	TB deaths
13	Bangladesh	156595	80000
36	China	1393337	41000
58	Ethiopia	94101	30000

77	India	1252140	240000
78	Indonesia	249866	64000
124	Nigeria	173615	160000
128	Pakistan	182143	49000
134	Philippines	98394	27000
141	Russian Federation	142834	17000
190	Viet Nam	91680	17000

These expressions can get long and complicated, making it easy to miss a crucial round or square bracket. In those cases it is best to break up the expression into small steps. The previous example could also be written as:

In []:

```
population = df['Population (1000s)']
deaths = df['TB deaths']
df[(population > 80000) & (deaths > 10000)]
```

Exercise 3 Bitwise operators

Complete Exercise 3 in the Exercise notebook 2.

1 Weather data

This week you will be looking at investigating historic weather data.

Figure 1

Of course, such data is hugely important for research into the large-scale, long-term shift in our planet's weather patterns and average temperatures – climate change. However, such data is also incredibly useful for more mundane planning purposes. To demonstrate the learning this week, I, Rob Griffiths, will be using historic weather data to try and plan a summer holiday in the UK. You'll use the data too and get a chance to work on your own project at the end of the week.

The dataset we'll use to do this will come from the Weather Underground, which creates weather forecasts from data sent to them by a worldwide network of over 100,000 weather enthusiasts who have personal weather stations on their house or in their garden.

In addition to creating weather forecasts from that data, the Weather Underground also keeps that data as historic weather records allowing members of the public to download weather datasets for a particular time period and location. These datasets are downloaded as CSV files, explained in the next step.

Datasets are rarely 'clean' and fit for purpose, so it will be necessary to clean up the data and 'mould it' for your purposes. You will then learn how to visualise data by creating graphs using the **`plot()`** function.

1.1 What is a CSV file?

A CSV file is a plain text file that is used to hold tabular data. The acronym CSV is short for 'comma-separated values'.

Figure 2

Take a look at the first few lines of a CSV file that holds the same data as the Excel file 'WHO POP TB all.xls' that you encountered in Week 2:

```
Country,Population (1000s),TB deaths
Afghanistan,30552,13000.0
Albania,3173,20.0
Algeria,39208,5100.0
Andorra,79,0.26
Angola,21472,6900.0
Antigua and Barbuda,90,1.2
Argentina,41446,570.0
Armenia,2977,170.0
```

Notice that the first line is a row of column names. The subsequent lines are rows of actual data that correspond to the column names. The row of column names is optional, but it is helpful in understanding the data in the following lines and making sure the right values fall in the right place. In this example, the first value on every row must be a string representing a country's name, the second value is an integer representing that country's population (in 1000s) and the third value is a decimal representing the number of deaths due to TB. Note that the third value is a decimal (like 0.26 deaths for Andorra) and not an integer because it is an estimate obtained from statistical processing of collected data.

Note that each value or column name is separated by a comma but actually any character can be used to separate values in a CSV file, including spaces and tabs etc., hence CSV can also stand for 'character-separated values'.

Because CSV files are in plain-text it makes the data easy to import into any spreadsheet program, database or pandas dataframe.

Before anything can be done with a CSV file with pandas, the following import statement must be executed:

In []:

```
from pandas import *
```

As you learned in Week 2, the import statement loads into memory all the code in the pandas module.

To read a CSV file into a dataframe, the pandas function **read_csv()** needs to be called.

In []:

```
df = read_csv('WHO POP TB all.csv')
```

The above code creates a dataframe from the data in the file **WHO POP TB all.csv** and assigns it to the variable **df**. This is the simplest usage of the **read_csv()** function, just using a single argument, a string that holds the name of the CSV file.

However the function can take many additional arguments (some of which you'll use later), which determine how the file is to be read.

In the next step, find out about dataframes and the 'dot' notation.

1.2 Dataframes and the 'dot' notation

In Week 2 you learned that dataframes have methods, which are like functions, that can only be called in the context of a dataframe.

For example, because the TB deaths dataframe **df** has a column named 'Country', the **sort_values()** method can be called like this:

In []:

```
df.sort_values('Country')
```

Because there is variable name, followed by a dot, followed by the method, this is called **dot notation**. Methods are said to be a property of a dataframe. In addition to methods, dataframes have another property – attributes.

Figure 3

Attributes

A dataframe attribute is like a variable that can only be accessed in the context of a dataframe. One such attribute is **`columns`** which holds a dataframe's column names.

So the expression **`df.columns`** evaluates to the value of the **`columns`** attribute inside the dataframe **`df`**. The following code will get and display the names of the columns in the dataframe **`df`**:

In []:

```
df.columns
```

Out[]:

```
Index(['Country', 'Population (1000s)', 'TB deaths'],
dtype='object')
```

1.3 Getting and displaying dataframe rows

Dataframes can have hundreds or thousands of rows, so it is not practical to display a whole dataframe.

However, there are a number of dataframe attributes and methods that allow you to get and display either a single row or a number of rows at a time. Three of the most useful methods are: **`iloc()`**, **`head()`** and **`tail()`**. Note that to distinguish methods and attributes, we write `()` after a method's name.

Figure 4

The iloc attribute

A dataframe has a default integer index for its rows, which starts at 0 (zero). You can get and display any single row in a dataframe by using the**`iloc`** attribute with the index of the row you want to access as its argument. For example, the following code will get and display the first row of data in the dataframe **`df`**, which is at index 0:

In []:

```
df.iloc[0]
```

Out[]:

```
Country Afghanistan
Population (1000s) 30552
TB deaths 13000
Name: 0, dtype: object
```

Similarly, the following code will get and display the third row of data in the dataframe **`df`**, which is at index 2:

In []:

```
df.iloc[2]
```

Out[]:

```
Country Algeria
Population (1000s) 39208
TB deaths 5100.0
Name: 0, dtype: object
```

The head() method

The first few rows of a dataframe can be printed out with the **`head()`** method.

You can tell **`head()`** is a method, rather than an attribute such as **`columns`**, because of the parentheses (round brackets) after the property name.

If you don't give any argument, i.e. don't put any number within those parentheses, the default behaviour is to return the first five rows of the dataframe. If you give an argument, it will print that number of rows (starting from the row indexed by 0).

For example, executing the following code will get and display the first five rows in the dataframe **`df`**.

In []:

```
df.head()
```

Out[]:

	Country	Population (1000s)	TB deaths
0	Afghanistan	30552	13000.00
1	Albania	3173	20.00
2	Algeria	39208	5100.00
3	Andorra	79	0.26
4	Angola	21472	6900.00

And, executing the following code will get and display the first seven rows in the dataframe **`df.`**

In []:

```
df.head(7)
```

Out[]:

	Country	Population (1000s)	TB deaths
0	Afghanistan	30552	13000.00
1	Albania	3173	20.00
2	Algeria	39208	5100.00
3	Andorra	79	0.26
4	Angola	21472	6900.00
5	Antigua and Barbuda	90	1.20
6	Argentina	41446	570.00

The tail() method

The **`tail()`** method is similar to the **`head()`** method.

If no argument is given, the last five rows of the dataframe are returned, otherwise the number of rows returned is dependent on the argument, just like for the **`head()`** method.

In []:

```
df.tail()
```

Out[]:

	Country	Population (1000s)	TB deaths
189	Venezuela (Bolivarian Republic of)	30405	480
190	Viet Nam	91680	17000
191	Yemen	24407	990
192	Zambia	14539	3600
193	Zimbabwe	14150	5700

1.4 Getting and displaying dataframe columns

You learned in Week 2 that you can get and display a single column of a dataframe by putting the name of the column (in quotes) within square brackets immediately after the dataframe's name.

For example, like this:

In []:

```
df['TB deaths']
```

You then get output like this:

Out[]:

```
0 13000.00
1 20.00
2  5100.00
3 0.26
4  6900.00
5 1.20
6 570.00
...
```

Notice that although there is an index, there is no column heading. This is because what is returned is not a new dataframe with a single column but an example of the **`Series`** data type.

Figure 5

Each column in a dataframe is an example of a series

The **`Series`** data type is a collection of values with an integer index that starts from zero. In addition, the **`Series`** data type has many of the same methods and attributes as the **`DataFrame`** data type, so you can still execute code like:

In []:

```
df['TB deaths'].head()
```

Out[]:

```
0 13000.00
1 20.00
```

```
2  5100.00
3 0.26
4  6900.00
Name: TB deaths, dtype: float64
```

And

In []:

```
df['TB deaths'].iloc[2]
```

Out[]:

```
5100.00
```

However, pandas does provide a mechanism for you to get and display one or more selected columns as a new dataframe in its own right. To do this you need to use a **list**. A list in Python consists of one or more items separated by commas and enclosed within square brackets, for example **`['Country']`** or **`['Country', 'Population (1000s)']`**. This list is then put within outer square brackets immediately after the dataframe's name, like this:

In []:

```
df[['Country']].head()
```

Out[]:

	Country
0	Afghanistan
1	Albania
2	Algeria
3	Andorra
4	Angola

Note that the column is now named. The expression **`df[['Country']]`**(with two square brackets) evaluates to a new dataframe (which happens to have a single column) rather than a series.

To get a new dataframe with multiple columns you just need to put more column names in the list, like this:

In []:

```
df[['Country', 'Population (1000s)']].head()
```

Out[]:

	Country	Population (1000s)
0	Afghanistan	30552
1	Albania	3173
2	Algeria	39208

3	Andorra	79
4	Angola	21472

The code has returned a new dataframe with just the `'Country'` and `'Population (1000s)’` columns.

Exercise 1 Dataframes and CSV files

Now that you’ve learned about CSV files and more about pandas you are ready to complete Exercise 1 in the exercise notebook 2.

Open the exercise 2 notebook and the data file you used last week WHO POP TB all.csv and save it in the folder you created in Week 1.

If you’re using Anaconda instead of CoCalc, remember that to open the notebook you’ll need to navigate to the notebook using Jupyter. Once it’s open, run the existing code in the notebook before you start the exercise. When you’ve completed the exercise, save the notebook. If you need a quick reminder of how to use Jupyter watch again the video in Week 1 Exercise 1.

1.5 Comparison operators

In Expressions, you learned that Python has arithmetic operators: +, /, - and * and that expressions such as 5 + 2 evaluate to a value (in this case the number 7).

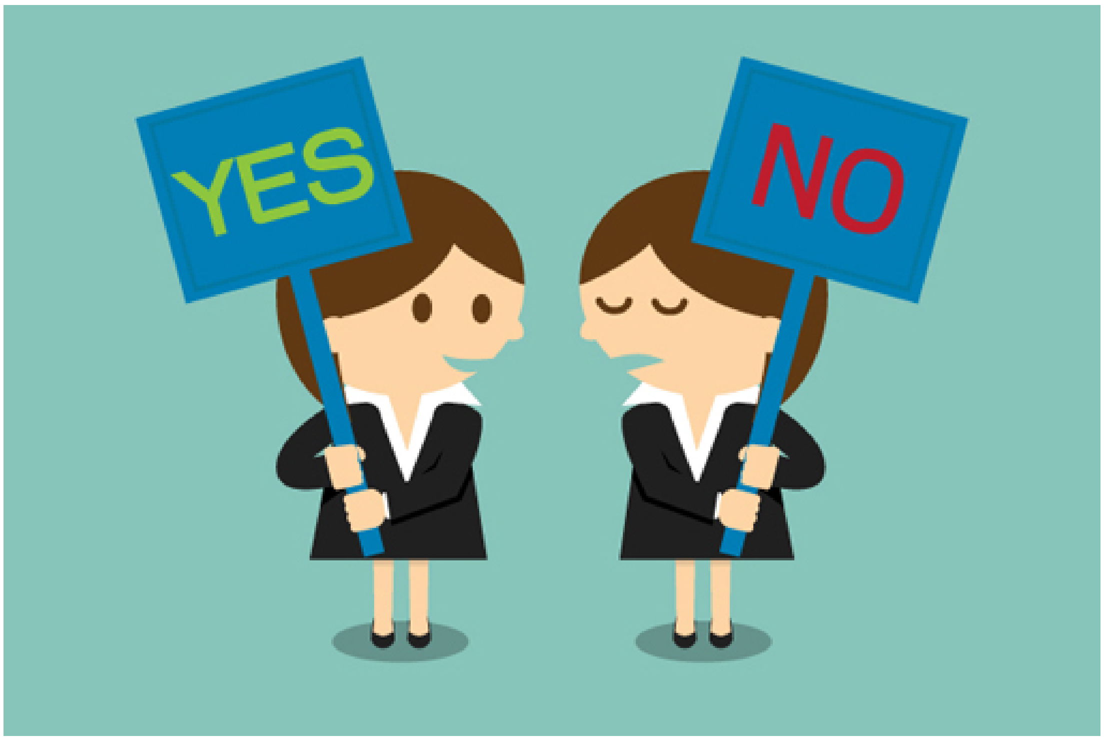


Figure 6

Python also has what are called comparison operators, these are:

```
== equals
!= not equal
< less than
> greater than
<= less than or equal to
>= greater than or equal to
```

Expressions involving these operators always evaluate to a Boolean value, that is **`True`** or **`False`**. Here are some examples:

```
2 = = 2 evaluates to True
2 + 2 = = 5 evaluates to False
2 != 1 + 1 evaluates to False
45 < 50 evaluates to True
20 > 30 evaluates to False
100 <= 100 evaluates to True
101 >= 100 evaluates to True
```

The comparison operators can be used with other types of data, not just numbers. Used with strings they compare using alphabetical order. For example:

`'aardvark' < 'zebra' evaluates to True`

In Calculating over columns you saw that when applied to whole columns, the arithmetic operators did the calculations row by row. Similarly, an expression like **`df['Country'] >= 'K'`** will compare the country names, row by row, against the string 'K' and record whether the result is **`True`** or **`False`** in a series like this:

```
0 False
1 False
2 False
3 False
4 False
5 False
...
Name: Country, dtype: bool
```

If such an expression is put within square brackets immediately after a dataframe's name, a new dataframe is obtained with only those rows where the result is **`True`**. So:

`df[df['Country'] >= 'K']`

returns a new dataframe with all the columns of **`df`** but with only the rows corresponding to countries starting with K or a letter later in the alphabet.

As another example, to see the data for countries with over 80 million inhabitants, the following code will return and display a new dataframe with all the columns of **`df`** but with only the rows where it is **`True`** that the value in the **`'Population (1000s)'`** column is greater than **`80000:`**

In []:

```
df[df['Population (1000s)'] > 80000]
```

Out[]:

	Country	Population (1000s)	TB deaths
13	Bangladesh	156595	80000
23	Brazil	200362	4400
36	China	1393337	41000
53	Egypt	82056	550
58	Ethiopia	94101	30000
65	Germany	82727	300
77	India	1252140	240000
78	Indonesia	249866	64000
85	Japan	127144	2100
109	Mexico	122332	2200
124	Nigeria	173615	160000
128	Pakistan	182143	49000
134	Philippines	98394	27000
141	Russian Federation	142834	17000
185	United States of America	320051	490
190	Viet Nam	91680	17000

Exercise 2 Comparison operators

You are ready to complete Exercise 2 in the Exercise notebook 2.

Remember to run the existing code in the notebook before you start the exercise. When you've completed the exercise, save the notebook.

1.6 Bitwise operators

To build more complicated expressions involving column comparisons, there are two bitwise operators.

Figure 7

The **&** operator means ‘and’ and the | operator (vertical bar, not uppercase letter ‘i’) means ‘or’. So, for example the expression:

```
(df['Country'] >= 'Latvia') & (df['Country'] <= 'Sweden')
```

will evaluate to a series containing Boolean values where the values are`True` only if the equivalent rows in the dataframe contain the countries ‘`Latvia`’ to ‘`Sweden`’, inclusive. However, the following expression which uses | (or) rather than & (and):

```
(df['Country'] >= 'Latvia') | (df['Country'] <= 'Sweden')
```

will evaluate to `True` for all countries, because every country comes alphabetically after ‘`Latvia`’ (e.g. the ‘UK’) or before '`Sweden`' (e.g. ‘`Brazil`’).

Note the round brackets around each comparison. Without them you will get an error.

The whole expression with multiple comparisons has to be put within `df[…]` to get a dataframe with only those rows that match the condition.

As a further example, using different columns, it is relatively easy to find the rows in `df` where '`Population (1000s)`' is greater than `80000` and where '`TB deaths`' are greater than `10000`.

In []:

```
df[(df['Population (1000s)'] > 80000) & (df['TB deaths'] > 10000)]
```

Out []:

	Country	Population (1000s)	TB deaths
13	Bangladesh	156595	80000
36	China	1393337	41000
58	Ethiopia	94101	30000

77	India	1252140	240000
78	Indonesia	249866	64000
124	Nigeria	173615	160000
128	Pakistan	182143	49000
134	Philippines	98394	27000
141	Russian Federation	142834	17000
190	Viet Nam	91680	17000

These expressions can get long and complicated, making it easy to miss a crucial round or square bracket. In those cases it is best to break up the expression into small steps. The previous example could also be written as:

In []:

```
population = df['Population (1000s)']
deaths = df['TB deaths']
df[(population > 80000) & (deaths > 10000)]
```

Exercise 3 Bitwise operators

Complete Exercise 3 in the Exercise notebook 2.

2 This week's quiz

Check what you've learned this week by taking the end-of-week quiz.

Week 3 practice quiz

Open the quiz in a new window or tab then come back here when you've finished.

3 Summary

This week looked at the importance of dataframes and the ‘dot’ notation, and the various dataframe methods. It also covered:

- CSV files
- Comparison operators
- Bitwise operators.

Next week looks at weather data and how to use the data to get answers to your questions.

Week 4: Cleaning up our act Part 2

1 Loading the weather data

You have learned some more about Python and the pandas module and tried it out on a fairly small dataset. You are now ready to explore a dataset from the Weather Underground.

Figure 1

Open the file London_2014.csv and save it in the disk folder or CoCalc project you created in Week 1.

Do not be tempted to open this file with Excel as this application will attempt to localise the data in the file, i.e. use your country's local data formats, which will make much of what follows rather incomprehensible! You can if you like open the file with a simple text editor, but **do not make any changes**.

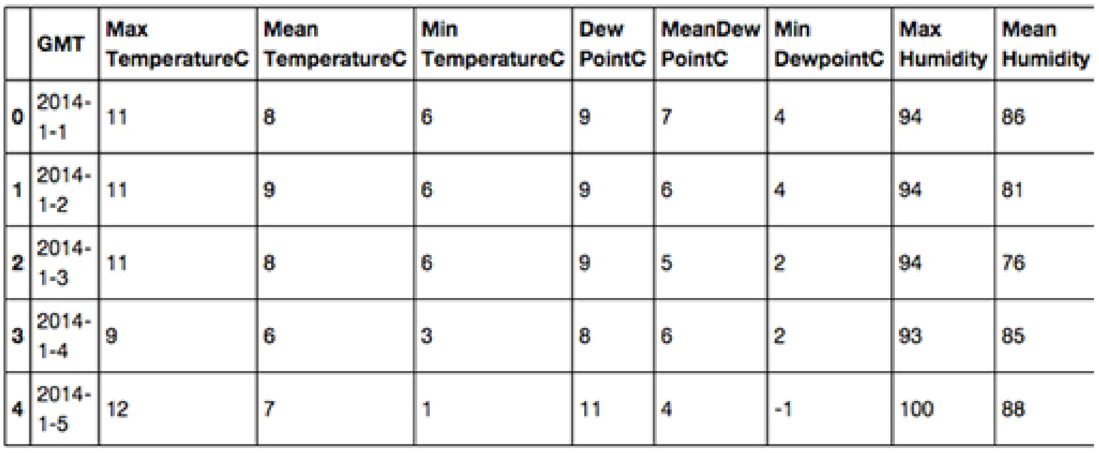
The CSV file can be loaded into a dataframe by executing the following code:

In []:

```
from pandas import *
london = read_csv('London_2014.csv')
```

```
london.head()
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC	Max Humidity	Mean Humidity
0	2014-1-1	11	8	6	9	7	4	94	86
1	2014-1-2	11	9	6	9	6	4	94	81
2	2014-1-3	11	8	6	9	5	2	94	76
3	2014-1-4	9	6	3	8	6	2	93	85
4	2014-1-5	12	7	1	11	4	-1	100	88

Figure 2

Note that the right hand side of the table has been cropped to fit on the page.

In the next section, you'll find out how to remove rogue spaces.

Important notice for learners outside of the EU

The Weather Underground automatically localises data based on from what country it detects you are accessing the web site. So, for example, if you are accessing the website from the USA wind speeds will be in MPH rather than km/h and temperatures in Fahrenheit rather than Celsius.

In order to change the settings so that the data is in European format you will need to click on the 'head and shoulders' icon on the top right of the Weather Underground web page and create a free Weather Underground account.

Once you have created an account, click on the 'cog' icon on the top right of the web page. Then:

- click on the C button to select Celsius
- click on 'More Settings' and select Units: metric
- click on 'Save My Preferences'.

Now, when you download the data, temperatures will be in Celsius and wind speeds in km/h etc.

1.1 Removing rogue spaces

One of the problems often encountered with CSV files is rogue spaces before or after data values or column names.

Figure 3

You learned earlier, in What is a CSV file? , that each value or column name is separated by a comma. However, if you opened 'London_2014.csv' in a text editor, you would see that in the row of column names sometimes there are spaces after a comma:

GMT,Max TemperatureC,Mean TemperatureC,Min TemperatureC,Dew PointC,MeanDew PointC,Min DewpointC,Max Humidity, Mean Humidity, Min Humidity, Max Sea Level PressurehPa, Mean Sea Level PressurehPa, Min Sea Level PressurehPa, Max VisibilityKm, Mean VisibilityKm, Min VisibilitykM, Max Wind SpeedKm/h, Mean Wind SpeedKm/h, Max Gust SpeedKm/h,Precipitationmm, CloudCover, Events,WindDirDegrees

For example, there is a space after the comma between **`Max Humidity`** and **`Mean Humidity`**. This means that when **`read_csv()`** reads the row of column names it will interpret a space after a comma as part of the next column name. So, for example, the column name after **`'Max Humidity'`** will be interpreted as **`' Mean Humidity'`** rather than what was intended, which is **`'Mean Humidity'`**. The ramification of this is that code such as:

```
london[['Mean Humidity']]
```

will cause a key error (see Selecting a column), as the column name is confusingly **`' Mean Humidity`** '.

This can easily be rectified by adding another argument to the **`read_csv()`** function:

```
skipinitialspace=True
```

which will tell **`read_csv()`** to ignore any spaces after a comma:

In []:

```
    london = read_csv('London_2014.csv', skipinitialspace=True)
```

The rogue spaces will no longer be in the dataframe and we can write code such as:

In []:

```
london[['Mean Humidity']].head()
```

Out[]:

	Mean Humidity
0	86
1	81
2	76
3	85
4	88

Note that a **`skipinitialspace=True`** argument won't remove a trailing space at the end of a column name.

Next, find out about extra characters and how to remove them.

1.2 Removing extra characters

If you opened London_2014.csv in a text editor once again and looked at the last column name you would see that the name is'WindDirDegrees
'.

What has happened here is that when the dataset was exported from the Weather Underground website an html line break **(**
) was added after the line of column headers which **`read_csv()`** has interpreted as the end part of the final column's name.

Figure 4

In fact, the problem is worse than this, let's look at some values in the final column:

In []:

```
london[['WindDirDegrees
']].head()
```

Out[]:

	WindDirDegrees
0	186
1	214
2	219
3	211
4	199

It's seems there is an html line break at the end of each line. If I opened 'London_2014.csv' in a text editor and looked at the ends of all lines in the file this would be confirmed.

Once again I'm not going to edit the CSV file but rather fix the problem in the dataframe. To change **`'WindDirDegrees`**
`'` to **`'WindDirDegrees'`** all I have to do is use the **`rename()`** method as follows:

In []:

```
london = london.rename(columns={'WindDirDegrees
':'WindDirDegrees'})
```

Don't worry about the syntax of the argument for **`rename()`** , just use this example as a template for whenever you need to change the name of a column.

Now I need to get rid of those pesky
html line breaks from the ends of the values in the **`'WindDirDegrees'`** column, so that they become something sensible. I can do that using the string method **`rstrip()`** which is used to remove characters from the end or 'rear' of a string, just like this:

In []:

```
london['WindDirDegrees'] = london['WindDirDegrees'].str.rstrip('
')
```

Again don't worry too much about the syntax of the code and simply use it as a template for whenever you need to process a whole column of values stripping characters from the end of each string value.

Let's display the first few rows of the ' **`WindDirDegrees`** ' to confirm the changes:

In []:

```
london[['WindDirDegrees']].head()
```

Out[]:

	WindDirDegrees
0	186
1	214
2	219
3	211
4	199

1.3 Missing values

As you heard in the video at the start of the week, missing values (also called null values) are one of the reasons to clean data.

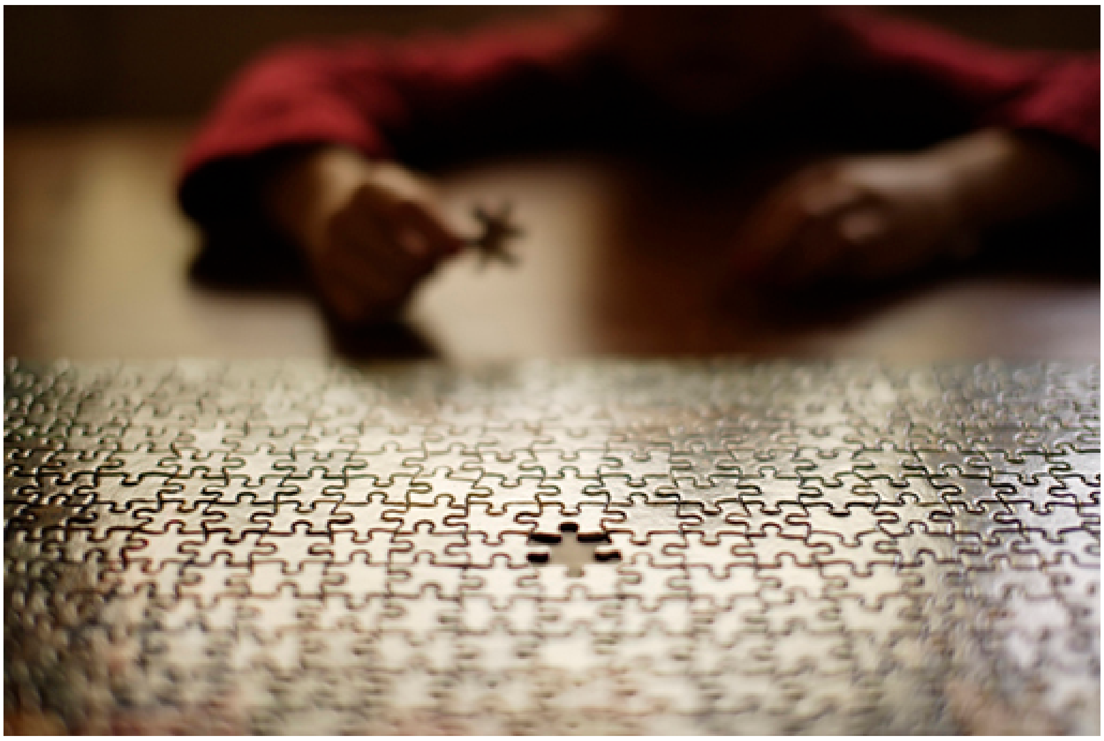


Figure 5

Finding missing values in a particular column can be done with the column method **`isnull()`** , like this:

In []:

```
london['Events'].isnull()
```

The above code returns a series of Boolean values, where **`True`** indicates that the corresponding row in the **`'Events'`** column is missing a value and **`False`** indicates the presence of a value. Here are the last few rows from the series:

```
...
360 False
```

```
361 True
362 True
363 True
364 False
Name: Events, dtype: bool
```

If, as you did with the comparison expressions, you put this code within square brackets after the dataframe's name, it will return a new dataframe consisting of all the rows without recorded events (rain, fog, thunderstorm, etc.):

In []:

```
london[london['Events'].isnull()]
```

As you will see in Exercise 4 of the exercise notebook, this will return a new dataframe with 114 rows, showing that more than one in three days had no particular event recorded. If you scroll the table to the right, you will see that all values in the `'Events'` column are marked `NaN` , which stands for 'Not a Number', but is also used to mark non-numeric missing values, like in this case (events are strings, not numbers).

Once you know how much and where data is missing, you have to decide what to do: ignore those rows? Replace with a fixed value? Replace with a computed value, like the mean?

In this case, only the first two options are possible. The method call `london.dropna()` will drop (remove) all rows that have a missing (non-available) value somewhere, returning a new dataframe. This will therefore also remove rows that have missing values in other columns.

The column method `fillna()` will replace all non-available values with the value given as argument. For this case, each NaN could be replaced by the empty string.

In []:

```
london['Events'] = london['Events'].fillna('')
london[london['Events'].isnull()]
```

The second line above will now show an empty dataframe, because there are no longer missing values in the events column.

As a final note on missing values, pandas ignores them when computing numeric statistics, i.e. you don't have to remove missing values before applying `sum(), median()` and other similar methods.

Learn about checking data types of each column in the next section.

1.4 Changing the value types of columns

The function `read_csv()` may, for many reasons, wrongly interpret the data type of the values in a column, so when cleaning data it's important to check the data types of each column are what is expected, and if necessary change them.

The data type of every column in a dataframe can be determined by looking at the dataframe's `dtypes` attribute, like this:

In []:

```
london.dtypes
```

Out[]:

```
GMT object
Max TemperatureC int64
```

```
Mean TemperatureC int64
Min TemperatureC int64
Dew PointC int64
MeanDew PointC int64
Min DewpointC int64
Max Humidity int64
Mean Humidity int64
Min Humidity int64
Max Sea Level PressurehPa int64
Mean Sea Level PressurehPa int64
Min Sea Level PressurehPa int64
Max VisibilityKm int64
Mean VisibilityKm int64
Min VisibilitykM int64
Max Wind SpeedKm/h int64
Mean Wind SpeedKm/h int64
Max Gust SpeedKm/h float64
Precipitationmm float64
CloudCover float64
Events object
WindDirDegrees object
dtype: object
```

In the above output, you can see the column names to the left and to the right the data types of the values in those columns.

- **`int64`** is the pandas data type for whole numbers such as **`55`** or **`2356`**
- **`float64`** is the pandas data type for decimal numbers such as **`55.25`** or **`2356.00`**
- **`object`** is the pandas data type for strings such as **`'hello world'`** or **`'rain'`**

Most of the column data types seem fine, however two are of concern, **`'GMT'`** and **`'WindDirDegrees'`** , both of which are of type **`object.`** Let's take a look at **`'WindDirDegrees'`** first.

Changing the data type of the `'WindDirDegrees'` column

The **`read_csv()`** method has interpreted the values in the **`'WindDirDegrees'`** column as strings (type **`object`**). This is because in the CSV file the values in that column had all been suffixed with that html line break string
so **`read_csv()`** had no alternative but to interpret the values as strings.

The values in the **`'WindDirDegrees'`** column are meant to represent wind direction in terms of degrees from true north (360) and meteorologists always define the wind direction as the direction the wind is coming from. So if you stand so that the wind is blowing directly into your face, the direction you are facing names the wind, so a westerly wind is reported as 270 degrees. The compass rose shown below should make this clearer:

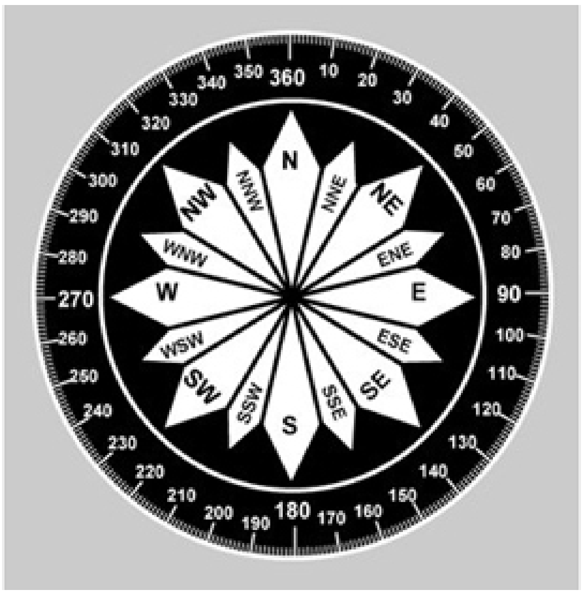


Figure 6 A compass rose

We need to be able to make queries such as ‘Get and display the rows where the wind direction is greater than 350 degrees’. To do this we need to change the data type of the ‘WindDirDegrees’ column from object to type **`int64`**. We can do that by using the **`astype()`** method like this:

`In []:`

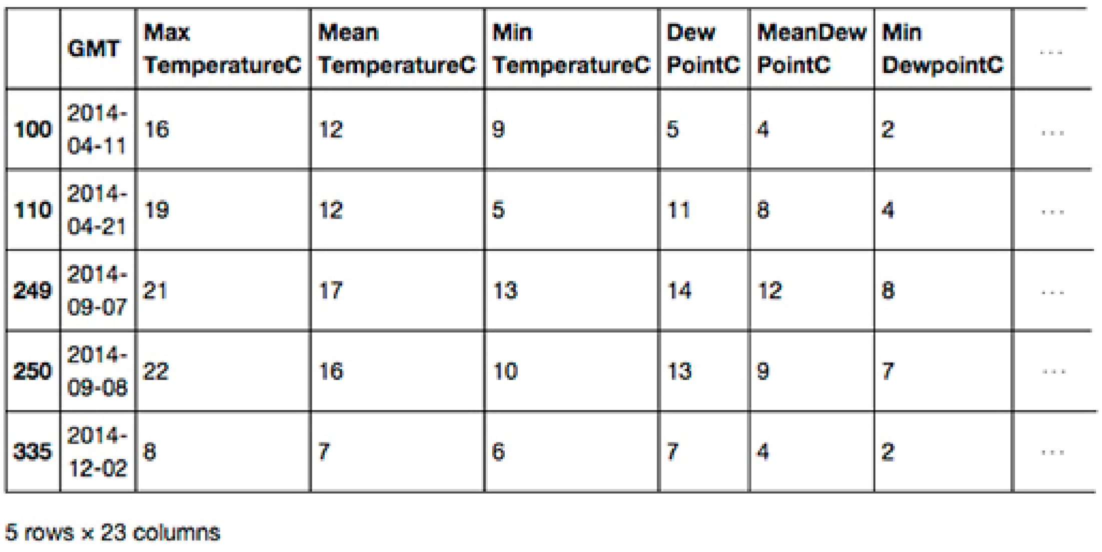
```
london['WindDirDegrees'] = london['WindDirDegrees'].astype('int64')
```

Now all the values in the **`'WindDirDegrees'`** column are of type **`int64`** and we can make our query:

`In []:`

```
london[london['WindDirDegrees'] > 350]
```

`Out[]:`



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC	...
100	2014-04-11	16	12	9	5	4	2	...
110	2014-04-21	19	12	5	11	8	4	...
249	2014-09-07	21	17	13	14	12	8	...
250	2014-09-08	22	16	10	13	9	7	...
335	2014-12-02	8	7	6	7	4	2	...

5 rows × 23 columns

Figure 7

*Note that the **`'WindDirDegrees'`** column is on the far right of the table and the right of the table has been cropped to fit on the page.*

Changing the data type of the 'GMT' column

Recall that I noted that the **'GMT'** column was of type **object** , the type pandas uses for strings.

The **'GMT'** column is supposed to represent dates. It would be helpful for the date values not to be strings to make it possible to make queries of the data such as 'Return the row where the date is 4 June 2014'.

Pandas has a function called **to_datetime()** which can convert a column of **object** (string) values such as those in the **'GMT'** column into values of a proper date type called **datetime64,** just like this:

In []:

```
london['GMT'] = to_datetime(london['GMT'])
#Then display the types of all the columns again so we
#can check the changes have been made.
london.dtypes
```

Out[]:

```
GMT datetime64[ns]
Max TemperatureC int64
Mean TemperatureC int64
Min TemperatureC int64
Dew PointC int64
MeanDew PointC int64
Min DewpointC int64
Max Humidity int64
Mean Humidity int64
Min Humidity int64
Max Sea Level PressurehPa int64
Mean Sea Level PressurehPa int64
Min Sea Level PressurehPa int64
Max VisibilityKm int64
Mean VisibilityKm int64
Min VisibilitykM int64
Max Wind SpeedKm/h int64
Mean Wind SpeedKm/h int64
Max Gust SpeedKm/h float64
Precipitationmm float64
CloudCover float64
Events object
WindDirDegrees int64
dtype: object
```

From the above output, we can confirm that the **'WindDirDegrees'** column type has been changed from **object** to **int64** and that the **'GMT'** column type has been changed from **object** to **datetime64**.

To make queries such as 'Return the row where the date is 4 June 2014' you'll need to be able to create a **datetime64** value to represent June 4 2014. It cannot be:

```
london[london['GMT'] == '2014-1-3']
```

because ‘2014-1-3’ is a string and the values in the ‘GMT’ column are of type **`datetime64`**. Instead you must create a **`datetime64`** value using **`thedatetime()`** function like this:

```
datetime(2014, 6, 4)
```

In the function call above, the first integer argument is the year, the second the month and the third the day.

First import the `datetime()` function from the similarly named `datetime` package by running the following line of code:

In []:

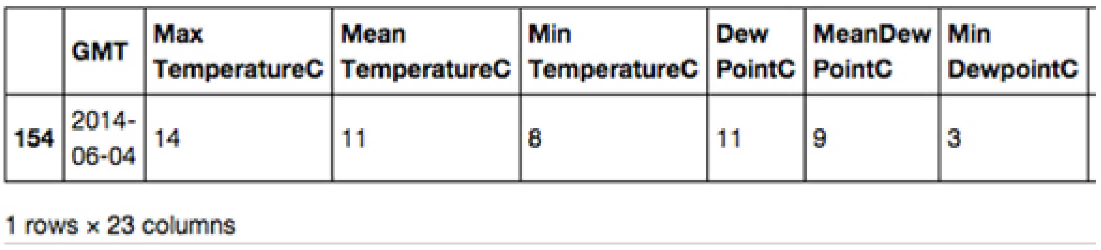
```
from datetime import datetime
```

Let’s try the function out by executing the code to ‘Return the row where the date is 4 June 2014’:

In []:

```
london[london['GMT'] == datetime(2014, 6, 4)]
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC
154	2014-06-04	14	11	8	11	9	3

1 rows × 23 columns

Figure 8

Note that the right side of the table has been cropped to fit on the page.

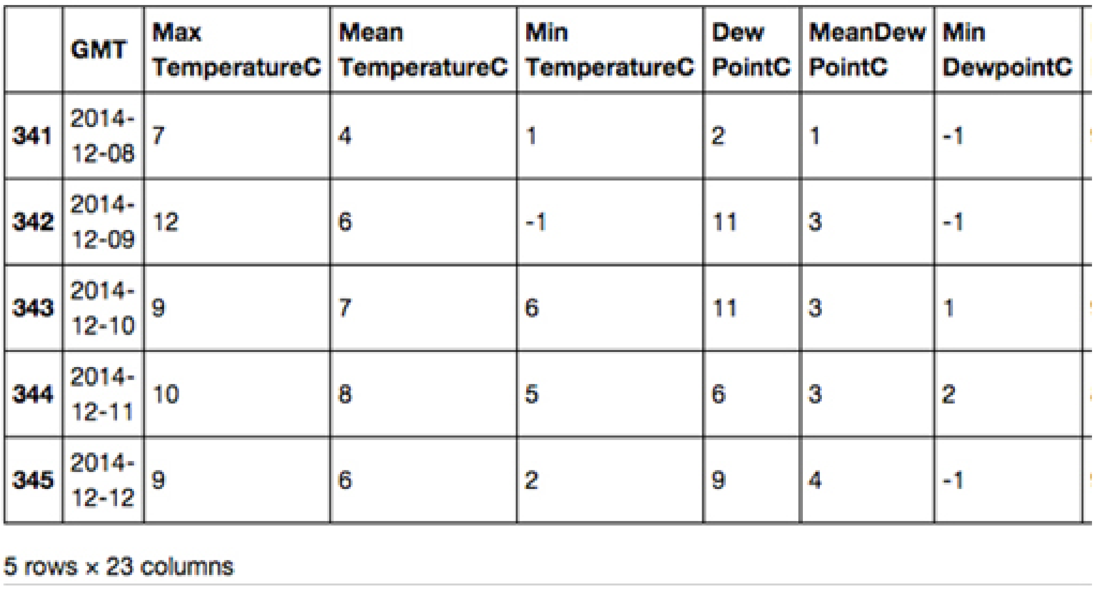
You can also now make more complex queries involving dates such as ‘Return all the rows where the date is between 8 December 2014 and 12 December 2014’, like this:

In []:

```
c

    london[(london['GMT'] >= datetime(2014, 12, 8))
    & (london['GMT'] <= datetime(2014, 12, 12))]
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC
341	2014-12-08	7	4	1	2	1	-1
342	2014-12-09	12	6	-1	11	3	-1
343	2014-12-10	9	7	6	11	3	1
344	2014-12-11	10	8	5	6	3	2
345	2014-12-12	9	6	2	9	4	-1

5 rows × 23 columns

Figure 9

Note that the right side of the table has been cropped to fit on the page.

Exercise 4 Display rows from dataframe

Now try Exercise 4 in the Exercise notebook 2.

If you're using Anaconda instead of CoCalc, remember that to open the notebook you'll need to navigate to the notebook using Jupyter.

Once the notebook is open, run the existing code in the notebook before you start the exercise. When you've completed the exercise, save the notebook. If you need a quick reminder of how to use Jupyter, watch again the video in Week 1 Exercise 1.

2 Every picture tells a story

It can be difficult and confusing to look at a table of rows of numbers and make any meaningful interpretation especially if there are many rows and columns.

Handily, pandas has a method called **`plot()`** which will visualise data for us by producing a chart.

Before using the **`plot()`** method, the following line of code must be executed (once) which tells Jupyter to display all charts inside this notebook, immediately after each call to **`plot()`:**

In []:

```
%matplotlib inline
```

To plot **'`Max Wind SpeedKm/h`** ', it's as simple as this code:

In []:

```
london['Max Wind SpeedKm/h'].plot(grid=True)
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x108340588>
```

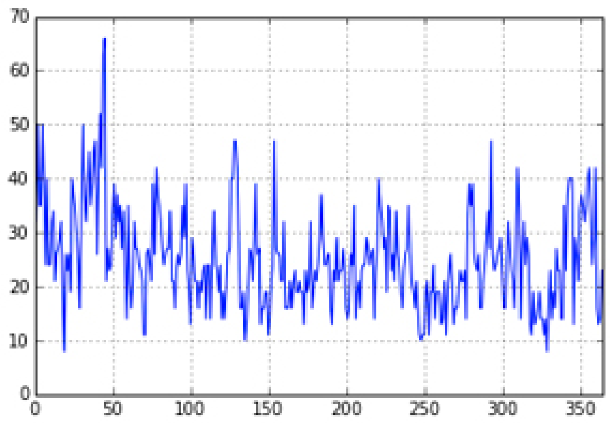


Figure 10

The **`grid=True`** argument makes the gridlines (the dotted lines in the image above) appear, which make values easier to read on the chart. The chart comes out a bit small, so you can make it bigger by giving the **`plot()`** method some extra information. The figsize units are inches.

In []:

```
london['Max Wind SpeedKm/h'].plot(grid=True, figsize=(10,5))
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10837a080>
```

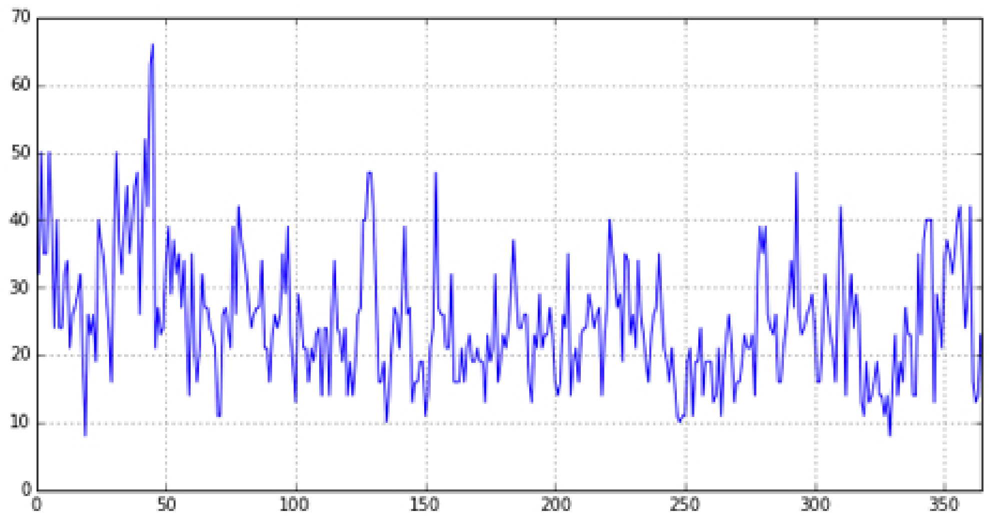


Figure 11

That's better! The argument given to the **`plot()`** method, **`figsize=(10,5)`** simply tells **`plot()`** that the x-axis should be 10 units wide and the y-axis should be 5 units high. In the above graph the x-axis (the numbers at the bottom) shows the dataframe's index, so 0 is 1 January and 50 is 18 February.

The y-axis (the numbers on the side) shows the range of wind speed in kilometres per hour. It is clear that the windiest day in 2014 was somewhere in mid-February and the wind reached about 66 kilometers per hour.

By default, the **`plot()`** method will try to generate a line, although as you'll see in a later week, it can produce other chart types too.

Exercise 5 Every picture tells a story

Now try Exercise 5 in the Exercise notebook 2.

If you're using Anaconda, remember that to open the notebook you'll need to navigate to the notebook using Jupyter.

2.1 Changing a dataframe's index

We have seen that by default every dataframe has an integer index for its rows which starts from 0.

The dataframe we've been using, **`london`** , has an index that goes from **`0`** to **`364`**. The row indexed by **`0`** holds data for the first day of the year and the row indexed by **`364`** holds data for the last day of the year. However, the column **`'GMT'`** holds **`datetime64`** values which would make a more intuitive index.

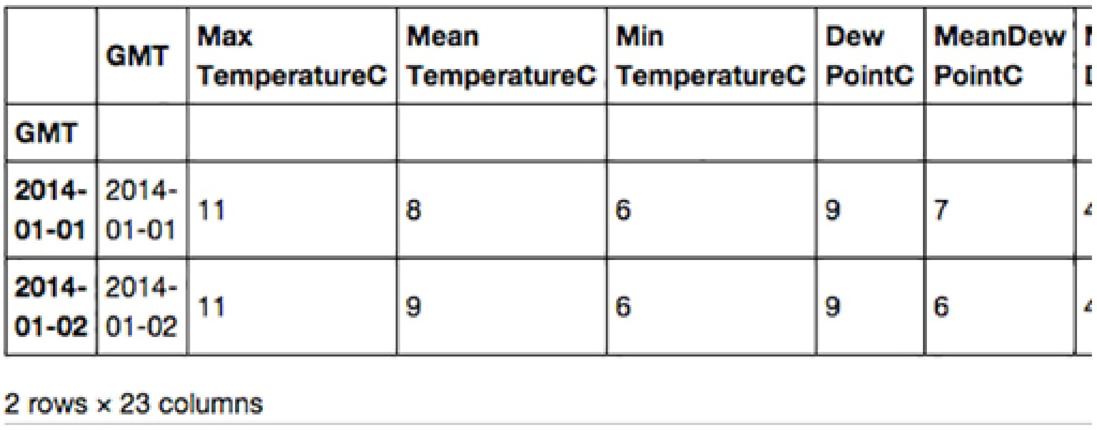
Changing the index to **`datetime64`** values is as easy as assigning to the dataframe's **`index`** attribute the contents of the **`'GMT'`** column, like this:

In []:

```
london.index = london['GMT']
```

```
#Display the first 2 rows
london.head(2)
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC
GMT						
2014-01-01	2014-01-01	11	8	6	9	7
2014-01-02	2014-01-02	11	9	6	9	6

2 rows × 23 columns

Figure 12

Note that the right of the table has been cropped to fit on the page.

Notice that the **`'GMT'`** column still remains and that the index has been labelled to show that it has been derived from the **`'GMT'`** column.

You can still access a row using the **`iloc`** attribute, so to get the first line in the dataframe you can simply execute:

In []:

```
london.iloc[0]
```

Out[]:

```
GMT 2014-01-01 00:00:00
Max TemperatureC 11
Mean TemperatureC 8
Min TemperatureC 6
Dew PointC 9
MeanDew PointC 7
Min DewpointC 4
Max Humidity 94
Mean Humidity 86
Min Humidity 73
Max Sea Level PressurehPa 1002
Mean Sea Level PressurehPa 993
Min Sea Level PressurehPa 984
Max VisibilityKm 31
Mean VisibilityKm 11
Min VisibilitykM 2
Max Wind SpeedKm/h 40
Mean Wind SpeedKm/h 26
Max Gust SpeedKm/h 66
Precipitationmm 9.91
```

```
CloudCover 4
Events Rain
WindDirDegrees 186
Name: 2014-01-01 00:00:00, dtype: object
```

But now you can now also use the **`datetime64`** index to get a row using the dataframe's **`loc`** attribute, like this:

In []:

```
london.loc[datetime(2014, 1, 1)]
```

Out[]:

```
GMT 2014-01-01 00:00:00
Max TemperatureC 11
Mean TemperatureC 8
Min TemperatureC 6
Dew PointC 9
MeanDew PointC 7
Min DewpointC 4
Max Humidity 94
Mean Humidity 86
Min Humidity 73
Max Sea Level PressurehPa 1002
Mean Sea Level PressurehPa 993
Min Sea Level PressurehPa 984
Max VisibilityKm 31
Mean VisibilityKm 11
Min VisibilitykM 2
Max Wind SpeedKm/h 40
Mean Wind SpeedKm/h 26
Max Gust SpeedKm/h 66
Precipitationmm 9.91
CloudCover 4
Events Rain
WindDirDegrees 186
Name: 2014-01-01 00:00:00, dtype: object
```

A query such as 'Return all the rows where the date is between 8 December and 12 December' which you did before (and can still do) with:

In []:

```
london[(london['GMT'] >= datetime(2014, 12, 8))
& (london['GMT'] <= datetime(2014, 12, 12))]
```

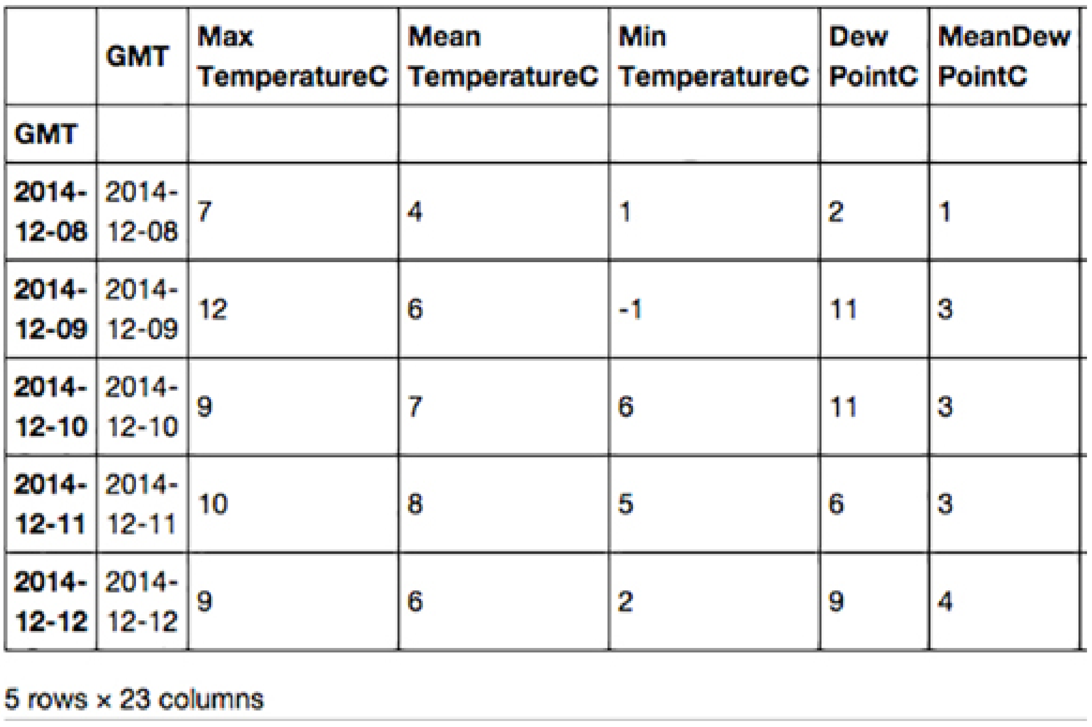
can now be done more succinctly like this:

In []:

```
london.loc[datetime(2014,12,8) : datetime(2014,12,12)]
#The meaning of the above code is get the rows between
#and including the indices datetime(2014,12,8) and
```

```
#datetime(2014,12,12)
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC
GMT						
2014-12-08	2014-12-08	7	4	1	2	1
2014-12-09	2014-12-09	12	6	-1	11	3
2014-12-10	2014-12-10	9	7	6	11	3
2014-12-11	2014-12-11	10	8	5	6	3
2014-12-12	2014-12-12	9	6	2	9	4

5 rows × 23 columns

Figure 13

Note that the right of the table has been cropped to fit on the page.

Because the table is in date order, we can be confident that only the rows with dates between 8 December 2014 and 12 December 2014 (inclusive) will be returned. However if the table had not been in date order, we would have needed to sort it first, like this:

`london = london.sort_index()`

Now there is a **`datetime64`** index, let's plot ' **`Max Wind SpeedKm/h`** 'again:

In []:

```
london['Max Wind SpeedKm/h'].plot(grid=True, figsize=(10,5))
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10829f7b8>
```

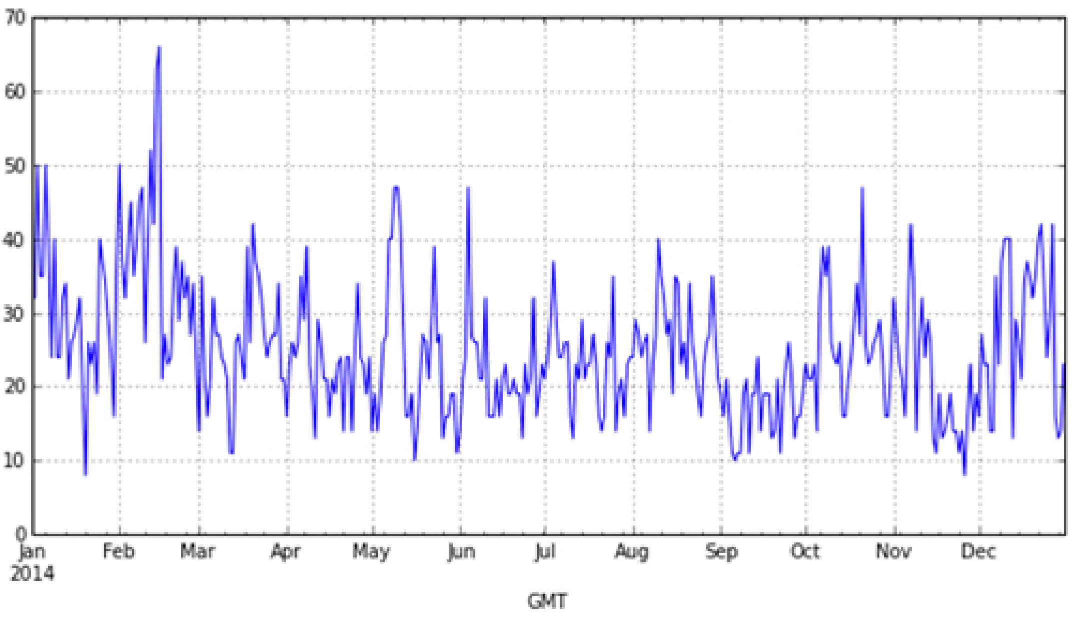


Figure 14

Now it is much clearer that the worst winds were in mid-February.

Exercise 6 Changing a dataframe's index

Now try Exercise 6 in the Exercise notebook 2.

2.2 The project

Your project this week is to find out what would have been the best two weeks of weather for a 2014 vacation in a capital of a BRICS country.

Figure 15

I've written up my analysis of the best two weeks of weather in London, UK, which you can open in project: 2: Holiday weather.

The structure is very simple: besides the introduction and the conclusions, there is one section for each step of the analysis – obtaining, cleaning and visualising the data.

Once you've worked through my analysis you should open a dataset for just one of the BRICS capitals: Brasilia, Moscow, Delhi, Beijing or Cape Town. The choice of capital is up to you. You should then work out the best two weeks, according to the weather, to choose for a two-week holiday in your chosen capital city.

Download the dataset for your chosen location as follows:

- Right click on the name of your chosen capital city above
- Choose to save the file via 'Download Linked File As...' Save the file with its default name to your downloads folder.
- If necessary, rename the file so that it has a .csv extension.
- Finally, move or copy te file to the disk folder or SageMathCloud by Cocalc project you created in Week 1.

Once again, **do not open the file with Excel** , but you could take a look using a text editor.

In my project, because I'm in London, which is often cold and rainy, I was looking for a two week period that had relatively high temperatures and little rain. If you choose a capital in a particularly hot and dry country you will probably be looking for relatively cool weather and low humidity.

Note that the London file has the dates in a column named 'GMT' whereas in the BRICS files they are in a column named 'Date'. You will need to change the Python code accordingly. You should also change the name of the variable, London, according to the capital you choose.

3 This week’s quiz

Now it’s time to complete the Week 4 badge quiz. It is similar to previous quizzes, but this time instead of answering five questions there will be fifteen.

Week 4 compulsory badge quiz

Remember, this quiz counts towards your badge. If you’re not successful the first time, you can attempt the quiz again in 24 hours.

4 Summary

Figure 16

This week you have learned how to:

- load a dataset into a dataframe from a CSV file
- clean data
- use the data to get answers to your questions.

Next week you will learn about the techniques behind the creation of a combined dataset.

You are now halfway through the course. The Open University would really appreciate your feedback and suggestions for future improvement in our optional end-of-course survey, which you will also have an opportunity to complete at the end of Week 8. Participation will be completely confidential and we will not pass on your details to others.

4.1 Week 4 glossary

Here is an alphabetical list of the terms introduced this week, for quick look-up.

Programming and data analysis concepts

The **bitwise operators** `&` (and) and `|` (or) are used in pandas to build more complicated expressions from two comparison expressions (typically involving column comparisons).

A **Boolean** has one of two possible values: `True` or `False`.

A **Comma Separated Values (CSV)** file is a plain text file that is used to hold tabular data.

A **list** is a sequence of values, separated by commas, and written within square brackets.

There are six **comparison operators** that can be used to compare number, string and date values. Expressions composed of these operators evaluate to `True` or `False`. These operators can also be used to compare every value in a column, row by row, against some number, string or date value. When used in this manner the operators return a series of Boolean values.

The **'dot' notation** is used to access a dataframe's methods and attributes.

The `Series` data type is a collection of values with an integer index that starts from zero. Each column in a dataframe is an example of the `Series` data type. The `Series` data type has many of the same methods as the `DataFrame` data type.

The `object` data type is how pandas represents strings.

The `datetime64` data type is how pandas represents dates.

The `int64` data type is how pandas represents integers (whole numbers).

The `float64` data type is how pandas represents floating point numbers (decimals).

Functions and methods

`asType(aType)` when applied to a dataframe column, the method changes the data type of each value in that column to the type given by the string `aType`.

`datetime(yyyy, mm, dd)` the function takes three arguments, `yyyy` a four digit integer representing a year, `mm` a two digit integer representing a month and `dd` a two digit integer representing a day. From these arguments the function creates and returns a value of `datetime64`.

`dropna()` when applied to a dataframe returns a new dataframe without the rows that have at least one missing value.

`head()` gets and displays the first five rows of a dataframe. Optionally the method can take an integer argument to specify how many rows (from and including row 0) to get and display.

`iloc[index]` gets and displays the row in the dataframe indicated by the integer argument `index`.

`isnull()` is a series method that checks which rows in that series have a missing value.

`fillna(value)` is a series method that returns a new series in which all missing values have been filled with the given value.

`plot()` when applied to a dataframe column of numeric values, the method displays a graph of those values. The x-axis shows the dataframe's index and the y-axis the range of the column's values. Before the method is called you first need to execute `%matplotlib inline`.

`read_csv(csvFile)` creates a dataframe from the dataset in the CSV file.

`rename(columns={oldName : newName})` renames the column `oldName` to `newName`.

`str.rstrip(suffix)` when applied to a dataframe column of string values, the method removes the argument **`suffix`** from the end of each string value in the column.

`tail()` gets and displays the last five rows of a dataframe. Optionally the method can take an integer argument to specify how many rows (until and including the last row) to get and display.

`to_datetime(aSeries)` when applied to a series, typically a column from a dataframe, this function returns a new series in which each value in **`aSeries`** has been changed to type **`datetime64`**.

Week 4: Cleaning up our act Part 2

1 Loading the weather data

You have learned some more about Python and the pandas module and tried it out on a fairly small dataset. You are now ready to explore a dataset from the Weather Underground.

Figure 1

Open the file London_2014.csv and save it in the disk folder or CoCalc project you created in Week 1.

Do not be tempted to open this file with Excel as this application will attempt to localise the data in the file, i.e. use your country's local data formats, which will make much of what follows rather incomprehensible! You can if you like open the file with a simple text editor, but **do not make any changes**.

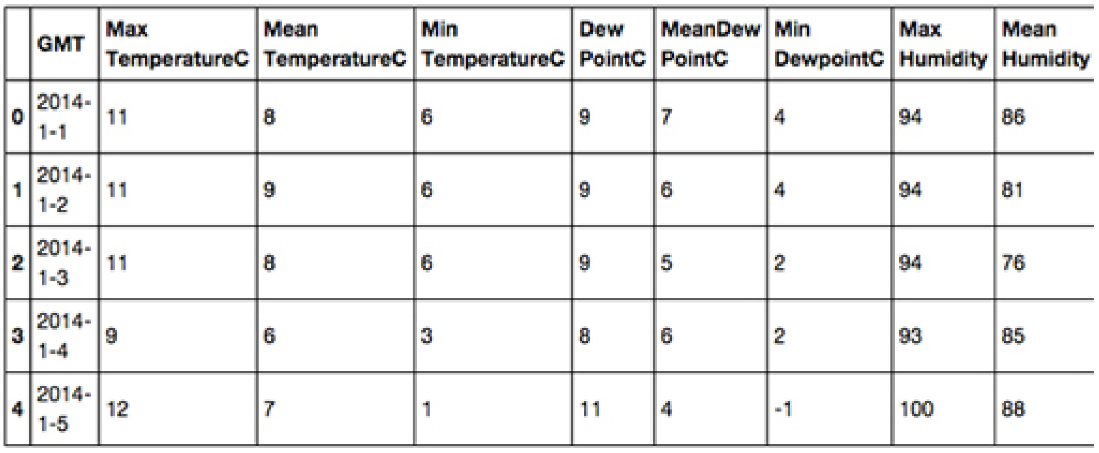
The CSV file can be loaded into a dataframe by executing the following code:

In []:

```
from pandas import *
london = read_csv('London_2014.csv')
```

```
london.head()
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC	Max Humidity	Mean Humidity
0	2014-1-1	11	8	6	9	7	4	94	86
1	2014-1-2	11	9	6	9	6	4	94	81
2	2014-1-3	11	8	6	9	5	2	94	76
3	2014-1-4	9	6	3	8	6	2	93	85
4	2014-1-5	12	7	1	11	4	-1	100	88

Figure 2

Note that the right hand side of the table has been cropped to fit on the page.

In the next section, you'll find out how to remove rogue spaces.

Important notice for learners outside of the EU

The Weather Underground automatically localises data based on from what country it detects you are accessing the web site. So, for example, if you are accessing the website from the USA wind speeds will be in MPH rather than km/h and temperatures in Fahrenheit rather than Celsius.

In order to change the settings so that the data is in European format you will need to click on the 'head and shoulders' icon on the top right of the Weather Underground web page and create a free Weather Underground account.

Once you have created an account, click on the 'cog' icon on the top right of the web page. Then:

- click on the C button to select Celsius
- click on 'More Settings' and select Units: metric
- click on 'Save My Preferences'.

Now, when you download the data, temperatures will be in Celsius and wind speeds in km/h etc.

1.1 Removing rogue spaces

One of the problems often encountered with CSV files is rogue spaces before or after data values or column names.

Figure 3

You learned earlier, in What is a CSV file? , that each value or column name is separated by a comma. However, if you opened ‘London_2014.csv’ in a text editor, you would see that in the row of column names sometimes there are spaces after a comma:

GMT,Max TemperatureC,Mean TemperatureC,Min TemperatureC,Dew PointC,MeanDew PointC,Min DewpointC,Max Humidity, Mean Humidity, Min Humidity, Max Sea Level PressurehPa, Mean Sea Level PressurehPa, Min Sea Level PressurehPa, Max VisibilityKm, Mean VisibilityKm, Min VisibilitykM, Max Wind SpeedKm/h, Mean Wind SpeedKm/h, Max Gust SpeedKm/h,Precipitationmm, CloudCover, Events,WindDirDe-grees

For example, there is a space after the comma between **`Max Humidity`** and **`Mean Humidity`**. This means that when **`read_csv()`** reads the row of column names it will interpret a space after a comma as part of the next column name. So, for example, the column name after **`'Max Humidity'`** will be interpreted as **`' Mean Humidity'`** rather than what was intended, which is **`'Mean Humidity'`**. The ramification of this is that code such as:

```
london[['Mean Humidity']]
```

will cause a key error (see Selecting a column), as the column name is confusingly **`' Mean Humidity`** '.

This can easily be rectified by adding another argument to the **`read_csv()`** function:

```
skipinitialspace=True
```

which will tell **`read_csv()`** to ignore any spaces after a comma:

In []:

```
    london = read_csv('London_2014.csv', skipinitialspace=True)
```

The rogue spaces will no longer be in the dataframe and we can write code such as:

In []:

```
london[['Mean Humidity']].head()
```

Out[]:

	Mean Humidity
0	86
1	81
2	76
3	85
4	88

Note that a `skipinitialspace=True` argument won't remove a trailing space at the end of a column name.

Next, find out about extra characters and how to remove them.

1.2 Removing extra characters

If you opened London_2014.csv in a text editor once again and looked at the last column name you would see that the name is'WindDirDegrees
'.

What has happened here is that when the dataset was exported from the Weather Underground website an html line break `(`
`)` was added after the line of column headers which `read_csv()` has interpreted as the end part of the final column's name.

Figure 4

In fact, the problem is worse than this, let's look at some values in the final column:

In []:

```
london[['WindDirDegrees
']].head()
```

Out[]:

	WindDirDegrees
0	186
1	214
2	219
3	211
4	199

It's seems there is an html line break at the end of each line. If I opened 'London_2014.csv' in a text editor and looked at the ends of all lines in the file this would be confirmed.

Once again I'm not going to edit the CSV file but rather fix the problem in the dataframe. To change **`'WindDirDegrees
'`** to **`'WindDirDegrees'`** all I have to do is use the **`rename()`** method as follows:

In []:

```
london = london.rename(columns={'WindDirDegrees
':'WindDirDegrees'})
```

Don't worry about the syntax of the argument for **`rename()`** , just use this example as a template for whenever you need to change the name of a column.

Now I need to get rid of those pesky
html line breaks from the ends of the values in the **`'WindDirDegrees'`** column, so that they become something sensible. I can do that using the string method **`rstrip()`** which is used to remove characters from the end or 'rear' of a string, just like this:

In []:

```
london['WindDirDegrees'] = london['WindDirDegrees'].str.rstrip('
')
```

Again don't worry too much about the syntax of the code and simply use it as a template for whenever you need to process a whole column of values stripping characters from the end of each string value.

Let's display the first few rows of the ' **`WindDirDegrees`** ' to confirm the changes:

In []:

```
london[['WindDirDegrees']].head()
```

Out[]:

	WindDirDegrees
0	186
1	214
2	219
3	211
4	199

1.3 Missing values

As you heard in the video at the start of the week, missing values (also called null values) are one of the reasons to clean data.

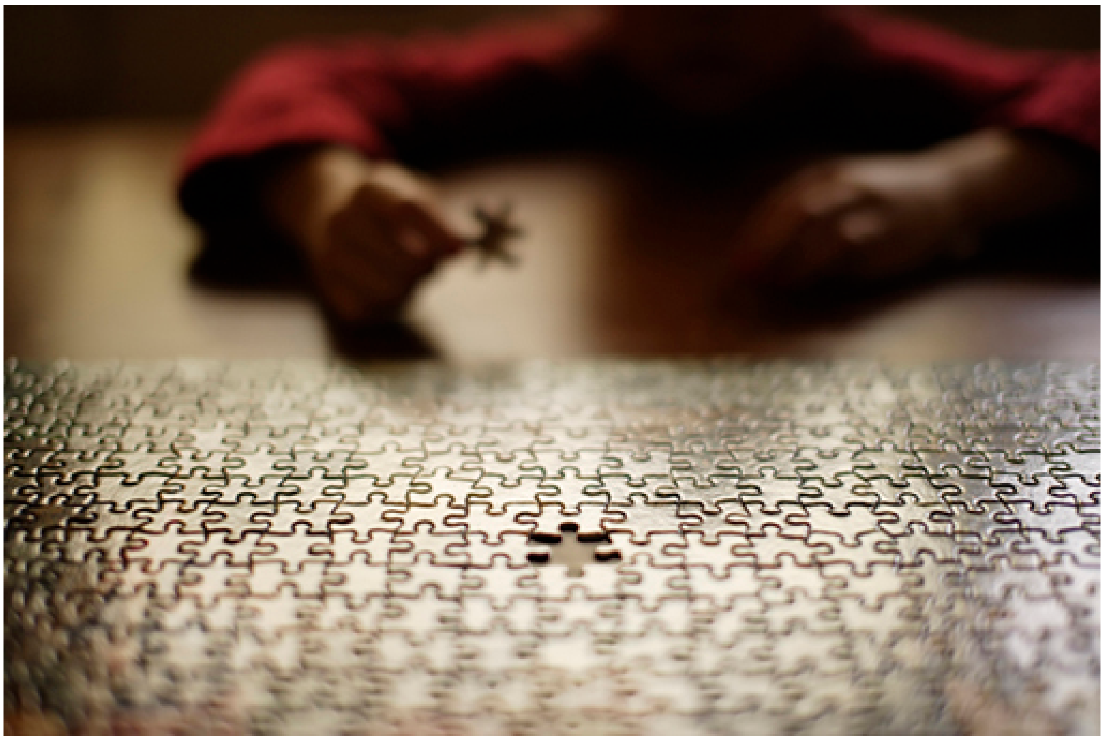


Figure 5

Finding missing values in a particular column can be done with the column method **`isnull()`** , like this:

In []:

```
london['Events'].isnull()
```

The above code returns a series of Boolean values, where **`True`** indicates that the corresponding row in the **`'Events'`** column is missing a value and **`False`** indicates the presence of a value. Here are the last few rows from the series:

```
...
360 False
```

```
361 True
362 True
363 True
364 False
Name: Events, dtype: bool
```

If, as you did with the comparison expressions, you put this code within square brackets after the dataframe's name, it will return a new dataframe consisting of all the rows without recorded events (rain, fog, thunderstorm, etc.):

In []:

```
london[london['Events'].isnull()]
```

As you will see in Exercise 4 of the exercise notebook, this will return a new dataframe with 114 rows, showing that more than one in three days had no particular event recorded. If you scroll the table to the right, you will see that all values in the **`'Events'`** column are marked **`NaN`** , which stands for 'Not a Number', but is also used to mark non-numeric missing values, like in this case (events are strings, not numbers).

Once you know how much and where data is missing, you have to decide what to do: ignore those rows? Replace with a fixed value? Replace with a computed value, like the mean?

In this case, only the first two options are possible. The method call **`london.dropna()`** will drop (remove) all rows that have a missing (non-available) value somewhere, returning a new dataframe. This will therefore also remove rows that have missing values in other columns.

The column method **`fillna()`** will replace all non-available values with the value given as argument. For this case, each NaN could be replaced by the empty string.

In []:

```
london['Events'] = london['Events'].fillna('')
london[london['Events'].isnull()]
```

The second line above will now show an empty dataframe, because there are no longer missing values in the events column.

As a final note on missing values, pandas ignores them when computing numeric statistics, i.e. you don't have to remove missing values before applying **`sum(), median()`** and other similar methods.

Learn about checking data types of each column in the next section.

1.4 Changing the value types of columns

The function **`read_csv()`** may, for many reasons, wrongly interpret the data type of the values in a column, so when cleaning data it's important to check the data types of each column are what is expected, and if necessary change them.

The data type of every column in a dataframe can be determined by looking at the dataframe's **`dtypes`** attribute, like this:

In []:

```
london.dtypes
```

Out[]:

```
GMT object
Max TemperatureC int64
```

```
Mean TemperatureC int64
Min TemperatureC int64
Dew PointC int64
MeanDew PointC int64
Min DewpointC int64
Max Humidity int64
Mean Humidity int64
Min Humidity int64
Max Sea Level PressurehPa int64
Mean Sea Level PressurehPa int64
Min Sea Level PressurehPa int64
Max VisibilityKm int64
Mean VisibilityKm int64
Min VisibilitykM int64
Max Wind SpeedKm/h int64
Mean Wind SpeedKm/h int64
Max Gust SpeedKm/h float64
Precipitationmm float64
CloudCover float64
Events object
WindDirDegrees object
dtype: object
```

In the above output, you can see the column names to the left and to the right the data types of the values in those columns.

- **`int64`** is the pandas data type for whole numbers such as **`55`** or **`2356`**
- **`float64`** is the pandas data type for decimal numbers such as **`55.25`** or **`2356.00`**
- **`object`** is the pandas data type for strings such as **`'hello world'`** or **`'rain'`**

Most of the column data types seem fine, however two are of concern, **`'GMT'`** and **`'WindDirDegrees'`** , both of which are of type **`object.`** Let's take a look at **`'WindDirDegrees'`** first.

Changing the data type of the `'WindDirDegrees'` column

The **`read_csv()`** method has interpreted the values in the **`'WindDirDegrees'`** column as strings (type **`object`**). This is because in the CSV file the values in that column had all been suffixed with that html line break string
so **`read_csv()`** had no alternative but to interpret the values as strings.

The values in the **`'WindDirDegrees'`** column are meant to represent wind direction in terms of degrees from true north (360) and meteorologists always define the wind direction as the direction the wind is coming from. So if you stand so that the wind is blowing directly into your face, the direction you are facing names the wind, so a westerly wind is reported as 270 degrees. The compass rose shown below should make this clearer:

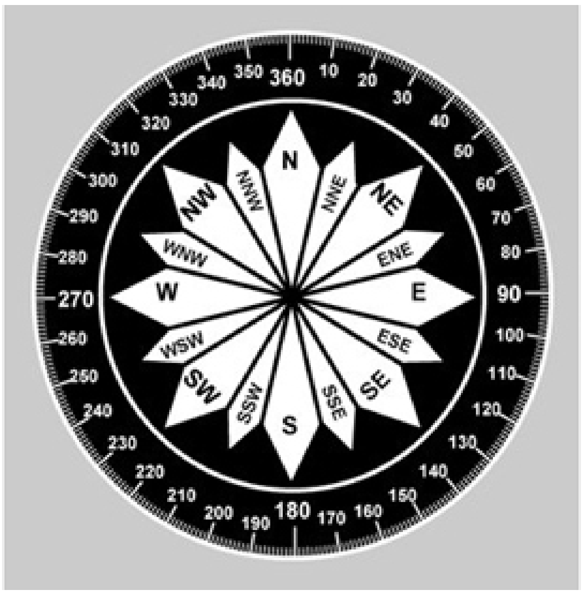


Figure 6 A compass rose

We need to be able to make queries such as ‘Get and display the rows where the wind direction is greater than 350 degrees’. To do this we need to change the data type of the ‘WindDirDegrees’ column from object to type **`int64`**. We can do that by using the **`astype()`** method like this:

In []:

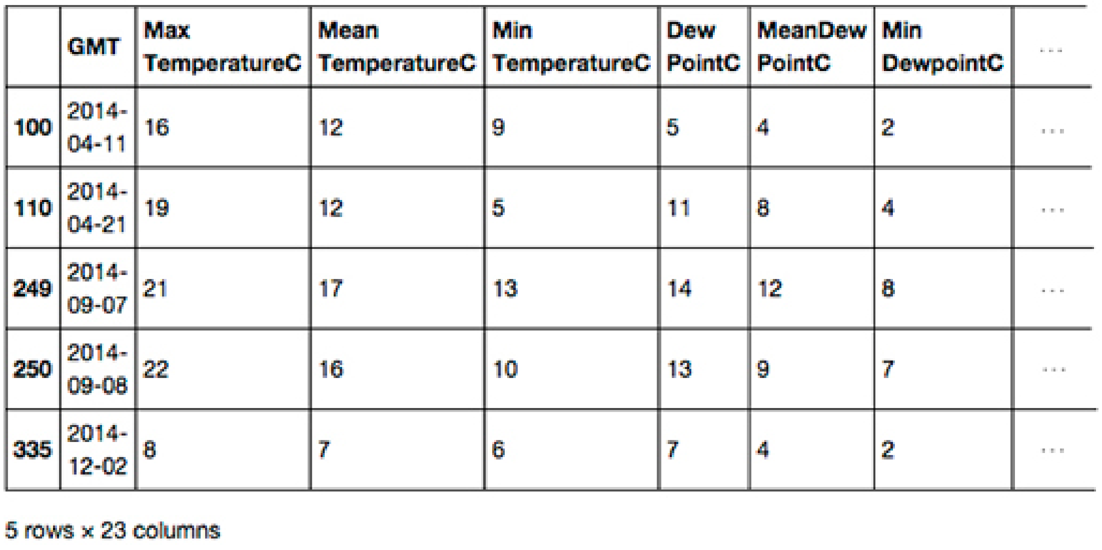
```
london['WindDirDegrees'] = london['WindDirDegrees'].astype('int64')
```

Now all the values in the **`'WindDirDegrees'`** column are of type **`int64`** and we can make our query:

In []:

```
london[london['WindDirDegrees'] > 350]
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC	...
100	2014-04-11	16	12	9	5	4	2	...
110	2014-04-21	19	12	5	11	8	4	...
249	2014-09-07	21	17	13	14	12	8	...
250	2014-09-08	22	16	10	13	9	7	...
335	2014-12-02	8	7	6	7	4	2	...

5 rows × 23 columns

Figure 7

*Note that the **`'WindDirDegrees'`** column is on the far right of the table and the right of the table has been cropped to fit on the page.*

Changing the data type of the ‘GMT’ column

Recall that I noted that the **'GMT'** column was of type **object** , the type pandas uses for strings.

The **'GMT'** column is supposed to represent dates. It would be helpful for the date values not to be strings to make it possible to make queries of the data such as ‘Return the row where the date is 4 June 2014’.

Pandas has a function called **to_datetime()** which can convert a column of **object** (string) values such as those in the **'GMT'** column into values of a proper date type called **datetime64,** just like this:

In []:

```
london['GMT'] = to_datetime(london['GMT'])
#Then display the types of all the columns again so we
#can check the changes have been made.
london.dtypes
```

Out[]:

```
GMT datetime64[ns]
Max TemperatureC int64
Mean TemperatureC int64
Min TemperatureC int64
Dew PointC int64
MeanDew PointC int64
Min DewpointC int64
Max Humidity int64
Mean Humidity int64
Min Humidity int64
Max Sea Level PressurehPa int64
Mean Sea Level PressurehPa int64
Min Sea Level PressurehPa int64
Max VisibilityKm int64
Mean VisibilityKm int64
Min VisibilitykM int64
Max Wind SpeedKm/h int64
Mean Wind SpeedKm/h int64
Max Gust SpeedKm/h float64
Precipitationmm float64
CloudCover float64
Events object
WindDirDegrees int64
dtype: object
```

From the above output, we can confirm that the **'WindDirDegrees'** column type has been changed from **object** to **int64** and that the **'GMT'** column type has been changed from **object** to **datetime64**.

To make queries such as ‘Return the row where the date is 4 June 2014’ you’ll need to be able to create a **datetime64** value to represent June 4 2014. It cannot be:

```
london[london['GMT'] == '2014-1-3']
```

because ‘2014-1-3’ is a string and the values in the ‘GMT’ column are of type **`datetime64`**. Instead you must create a **`datetime64`** value using **`thedatetime()`** function like this:

```
datetime(2014, 6, 4)
```

In the function call above, the first integer argument is the year, the second the month and the third the day.

First import the `datetime()` function from the similarly named `datetime` package by running the following line of code:

In []:

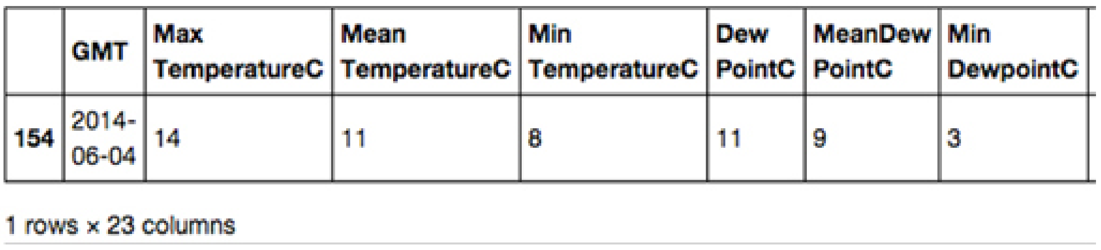
```
from datetime import datetime
```

Let’s try the function out by executing the code to ‘Return the row where the date is 4 June 2014’:

In []:

```
london[london['GMT'] == datetime(2014, 6, 4)]
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC
154	2014-06-04	14	11	8	11	9	3

1 rows × 23 columns

Figure 8

Note that the right side of the table has been cropped to fit on the page.

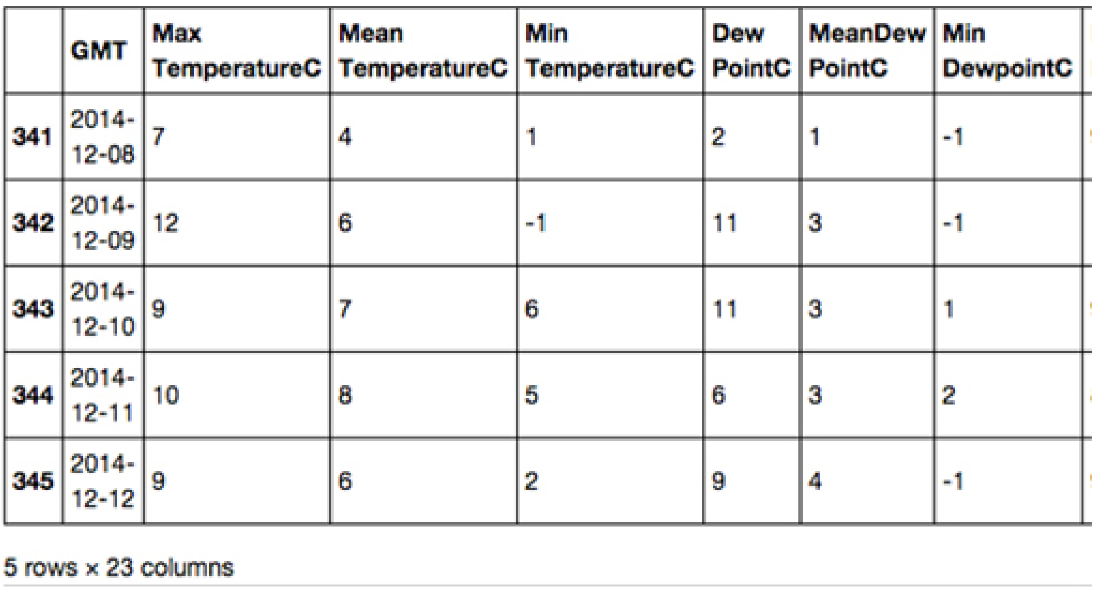
You can also now make more complex queries involving dates such as ‘Return all the rows where the date is between 8 December 2014 and 12 December 2014’, like this:

In []:

```
c

    london[(london['GMT'] >= datetime(2014, 12, 8))
    & (london['GMT'] <= datetime(2014, 12, 12))]
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC	Min DewpointC
341	2014-12-08	7	4	1	2	1	-1
342	2014-12-09	12	6	-1	11	3	-1
343	2014-12-10	9	7	6	11	3	1
344	2014-12-11	10	8	5	6	3	2
345	2014-12-12	9	6	2	9	4	-1

5 rows × 23 columns

Figure 9

Note that the right side of the table has been cropped to fit on the page.

Exercise 4 Display rows from dataframe

Now try Exercise 4 in the Exercise notebook 2.

If you're using Anaconda instead of CoCalc, remember that to open the notebook you'll need to navigate to the notebook using Jupyter.

Once the notebook is open, run the existing code in the notebook before you start the exercise. When you've completed the exercise, save the notebook. If you need a quick reminder of how to use Jupyter, watch again the video in Week 1 Exercise 1.

2 Every picture tells a story

It can be difficult and confusing to look at a table of rows of numbers and make any meaningful interpretation especially if there are many rows and columns.

Handily, pandas has a method called **`plot()`** which will visualise data for us by producing a chart.

Before using the **`plot()`** method, the following line of code must be executed (once) which tells Jupyter to display all charts inside this notebook, immediately after each call to **`plot()`:**

In []:

```
%matplotlib inline
```

To plot **‘`Max Wind SpeedKm/h`** ’, it’s as simple as this code:

In []:

```
london['Max Wind SpeedKm/h'].plot(grid=True)
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x108340588>
```

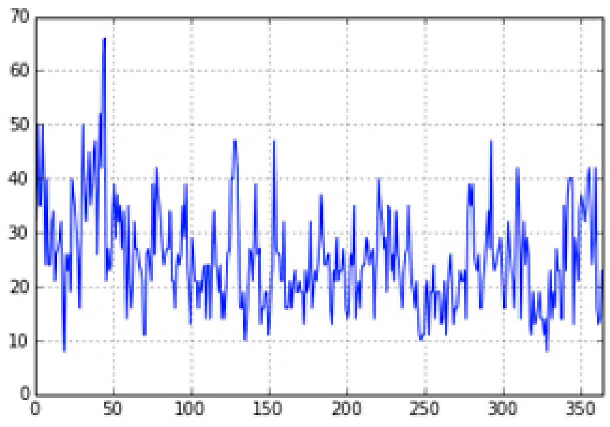


Figure 10

The **`grid=True`** argument makes the gridlines (the dotted lines in the image above) appear, which make values easier to read on the chart. The chart comes out a bit small, so you can make it bigger by giving the **`plot()`** method some extra information. The figsize units are inches.

In []:

```
london['Max Wind SpeedKm/h'].plot(grid=True, figsize=(10,5))
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10837a080>
```

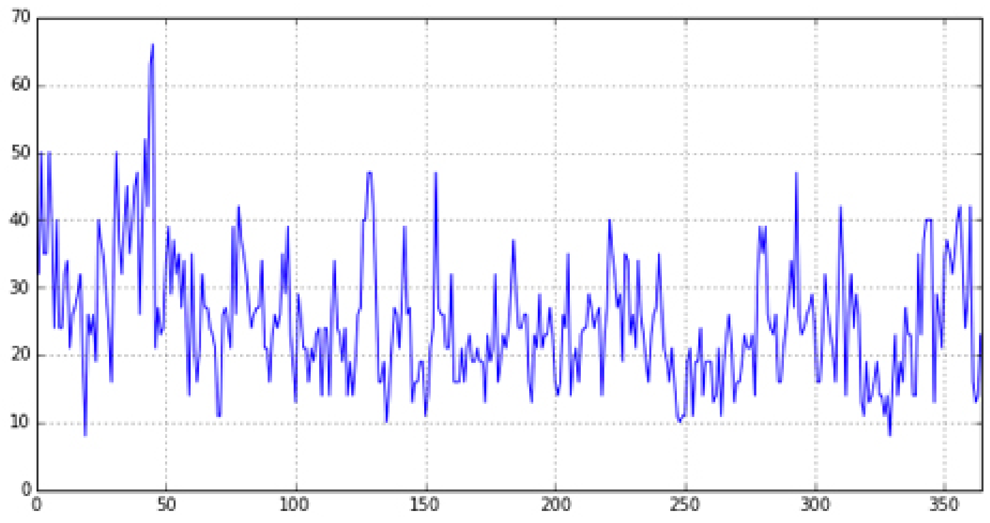


Figure 11

That’s better! The argument given to the **`plot()`** method, **`figsize=(10,5)`** simply tells **`plot()`** that the x-axis should be 10 units wide and the y-axis should be 5 units high. In the above graph the x-axis (the numbers at the bottom) shows the dataframe’s index, so 0 is 1 January and 50 is 18 February.

The y-axis (the numbers on the side) shows the range of wind speed in kilometres per hour. It is clear that the windiest day in 2014 was somewhere in mid-February and the wind reached about 66 kilometers per hour.

By default, the **`plot()`** method will try to generate a line, although as you’ll see in a later week, it can produce other chart types too.

Exercise 5 Every picture tells a story

Now try Exercise 5 in the Exercise notebook 2.

If you’re using Anaconda, remember that to open the notebook you’ll need to navigate to the notebook using Jupyter.

2.1 Changing a dataframe’s index

We have seen that by default every dataframe has an integer index for its rows which starts from 0.

The dataframe we’ve been using, **`london`** , has an index that goes from **`0`** to **`364`**. The row indexed by **`0`** holds data for the first day of the year and the row indexed by **`364`** holds data for the last day of the year. However, the column **`'GMT'`** holds **`datetime64`** values which would make a more intuitive index.

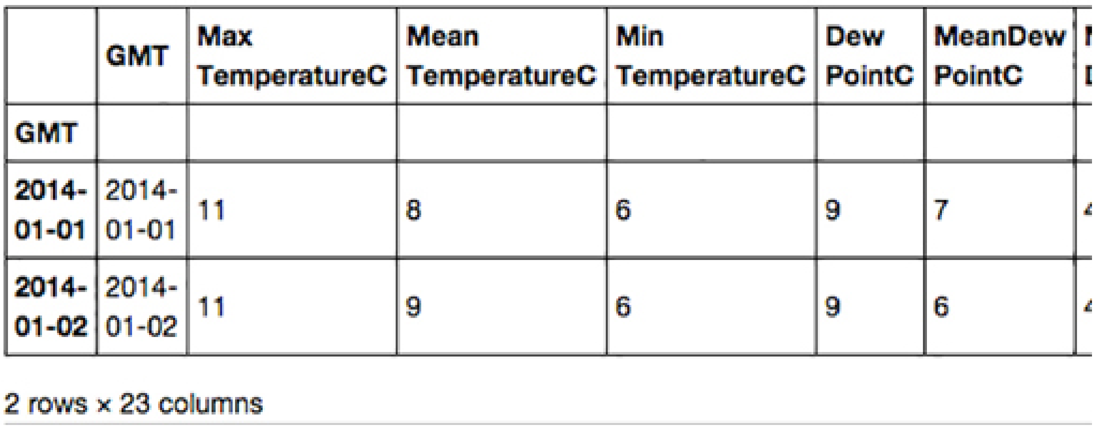
Changing the index to **`datetime64`** values is as easy as assigning to the dataframe’s **`index`** attribute the contents of the **`'GMT'`** column, like this:

In []:

```
london.index = london['GMT']
```

```
#Display the first 2 rows
london.head(2)
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC
GMT						
2014-01-01	2014-01-01	11	8	6	9	7
2014-01-02	2014-01-02	11	9	6	9	6

2 rows × 23 columns

Figure 12

Note that the right of the table has been cropped to fit on the page.

Notice that the **'GMT'** column still remains and that the index has been labelled to show that it has been derived from the **'GMT'** column.

You can still access a row using the **iloc** attribute, so to get the first line in the dataframe you can simply execute:

In []:

```
london.iloc[0]
```

Out[]:

```
GMT 2014-01-01 00:00:00
Max TemperatureC 11
Mean TemperatureC 8
Min TemperatureC 6
Dew PointC 9
MeanDew PointC 7
Min DewpointC 4
Max Humidity 94
Mean Humidity 86
Min Humidity 73
Max Sea Level PressurehPa 1002
Mean Sea Level PressurehPa 993
Min Sea Level PressurehPa 984
Max VisibilityKm 31
Mean VisibilityKm 11
Min VisibilitykM 2
Max Wind SpeedKm/h 40
Mean Wind SpeedKm/h 26
Max Gust SpeedKm/h 66
Precipitationmm 9.91
```

```
CloudCover 4
Events Rain
WindDirDegrees 186
Name: 2014-01-01 00:00:00, dtype: object
```

But now you can now also use the **`datetime64`** index to get a row using the dataframe's **`loc`** attribute, like this:

In []:

```
london.loc[datetime(2014, 1, 1)]
```

Out[]:

```
GMT 2014-01-01 00:00:00
Max TemperatureC 11
Mean TemperatureC 8
Min TemperatureC 6
Dew PointC 9
MeanDew PointC 7
Min DewpointC 4
Max Humidity 94
Mean Humidity 86
Min Humidity 73
Max Sea Level PressurehPa 1002
Mean Sea Level PressurehPa 993
Min Sea Level PressurehPa 984
Max VisibilityKm 31
Mean VisibilityKm 11
Min VisibilitykM 2
Max Wind SpeedKm/h 40
Mean Wind SpeedKm/h 26
Max Gust SpeedKm/h 66
Precipitationmm 9.91
CloudCover 4
Events Rain
WindDirDegrees 186
Name: 2014-01-01 00:00:00, dtype: object
```

A query such as 'Return all the rows where the date is between 8 December and 12 December' which you did before (and can still do) with:

In []:

```
london[(london['GMT'] >= datetime(2014, 12, 8))
& (london['GMT'] <= datetime(2014, 12, 12))]
```

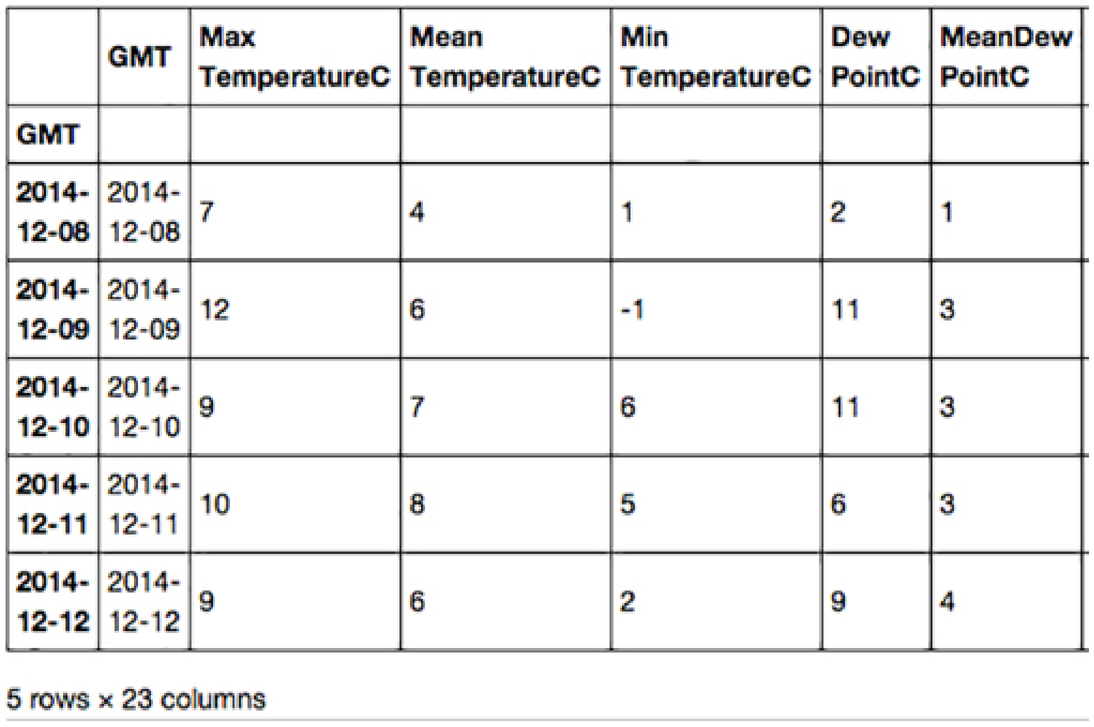
can now be done more succinctly like this:

In []:

```
london.loc[datetime(2014,12,8) : datetime(2014,12,12)]
#The meaning of the above code is get the rows between
#and including the indices datetime(2014,12,8) and
```

```
#datetime(2014,12,12)
```

Out[]:



	GMT	Max TemperatureC	Mean TemperatureC	Min TemperatureC	Dew PointC	MeanDew PointC
GMT						
2014-12-08	2014-12-08	7	4	1	2	1
2014-12-09	2014-12-09	12	6	-1	11	3
2014-12-10	2014-12-10	9	7	6	11	3
2014-12-11	2014-12-11	10	8	5	6	3
2014-12-12	2014-12-12	9	6	2	9	4

5 rows × 23 columns

Figure 13

Note that the right of the table has been cropped to fit on the page.

Because the table is in date order, we can be confident that only the rows with dates between 8 December 2014 and 12 December 2014 (inclusive) will be returned. However if the table had not been in date order, we would have needed to sort it first, like this:

`london = london.sort_index()`

Now there is a **`datetime64`** index, let’s plot ' **`Max Wind SpeedKm/h`** 'again:

In []:

```
london['Max Wind SpeedKm/h'].plot(grid=True, figsize=(10,5))
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10829f7b8>
```

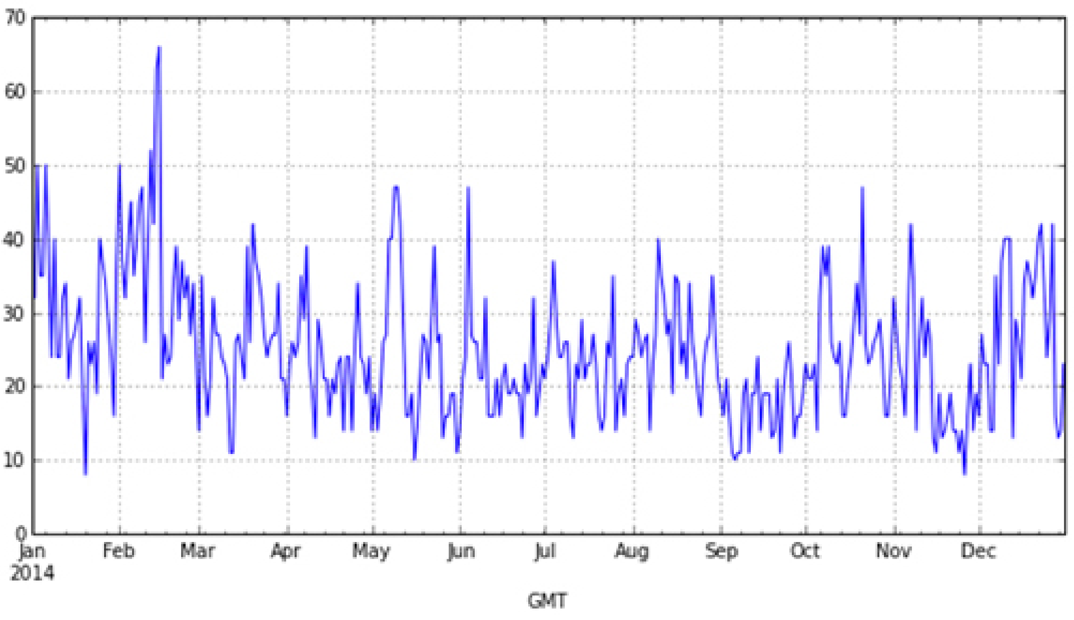


Figure 14

Now it is much clearer that the worst winds were in mid-February.

Exercise 6 Changing a dataframe's index

Now try Exercise 6 in the Exercise notebook 2.

2.2 The project

Your project this week is to find out what would have been the best two weeks of weather for a 2014 vacation in a capital of a BRICS country.

Figure 15

I've written up my analysis of the best two weeks of weather in London, UK, which you can open in project: 2: Holiday weather.

The structure is very simple: besides the introduction and the conclusions, there is one section for each step of the analysis – obtaining, cleaning and visualising the data.

Once you've worked through my analysis you should open a dataset for just one of the BRICS capitals: Brasilia, Moscow, Delhi, Beijing or Cape Town. The choice of capital is up to you. You should then work out the best two weeks, according to the weather, to choose for a two-week holiday in your chosen capital city.

Download the dataset for your chosen location as follows:

- Right click on the name of your chosen capital city above
- Choose to save the file via 'Download Linked File As...' Save the file with its default name to your downloads folder.
- If necessary, rename the file so that it has a .csv extension.
- Finally, move or copy te file to the disk folder or SageMathCloud by Cocalc project you created in Week 1.

Once again, **do not open the file with Excel** , but you could take a look using a text editor.

In my project, because I'm in London, which is often cold and rainy, I was looking for a two week period that had relatively high temperatures and little rain. If you choose a capital in a particularly hot and dry country you will probably be looking for relatively cool weather and low humidity.

Note that the London file has the dates in a column named 'GMT' whereas in the BRICS files they are in a column named 'Date'. You will need to change the Python code accordingly. You should also change the name of the variable, London, according to the capital you choose.

3 This week's quiz

Now it's time to complete the Week 4 badge quiz. It is similar to previous quizzes, but this time instead of answering five questions there will be fifteen.

Week 4 compulsory badge quiz

Remember, this quiz counts towards your badge. If you're not successful the first time, you can attempt the quiz again in 24 hours.

4 Summary

Figure 16

This week you have learned how to:

- load a dataset into a dataframe from a CSV file
- clean data
- use the data to get answers to your questions.

Next week you will learn about the techniques behind the creation of a combined dataset.

You are now halfway through the course. The Open University would really appreciate your feedback and suggestions for future improvement in our optional end-of-course survey, which you will also have an opportunity to complete at the end of Week 8. Participation will be completely confidential and we will not pass on your details to others.

4.1 Week 4 glossary

Here is an alphabetical list of the terms introduced this week, for quick look-up.

Programming and data analysis concepts

The **bitwise operators** `&` (and) and `|` (or) are used in pandas to build more complicated expressions from two comparison expressions (typically involving column comparisons).

A **Boolean** has one of two possible values: `True` or `False`.

A **Comma Separated Values (CSV)** file is a plain text file that is used to hold tabular data.

A **list** is a sequence of values, separated by commas, and written within square brackets.

There are six **comparison operators** that can be used to compare number, string and date values. Expressions composed of these operators evaluate to `True` or `False`. These operators can also be used to compare every value in a column, row by row, against some number, string or date value. When used in this manner the operators return a series of Boolean values.

The **'dot' notation** is used to access a dataframe's methods and attributes.

The `Series` data type is a collection of values with an integer index that starts from zero. Each column in a dataframe is an example of the `Series` data type. The `Series` data type has many of the same methods as the `DataFrame` data type.

The `object` data type is how pandas represents strings.

The `datetime64` data type is how pandas represents dates.

The `int64` data type is how pandas represents integers (whole numbers).

The `float64` data type is how pandas represents floating point numbers (decimals).

Functions and methods

`asType(aType)` when applied to a dataframe column, the method changes the data type of each value in that column to the type given by the string `aType`.

`datetime(yyyy, mm, dd)` the function takes three arguments, `yyyy` a four digit integer representing a year, `mm` a two digit integer representing a month and `dd` a two digit integer representing a day. From these arguments the function creates and returns a value of `datetime64`.

`dropna()` when applied to a dataframe returns a new dataframe without the rows that have at least one missing value.

`head()` gets and displays the first five rows of a dataframe. Optionally the method can take an integer argument to specify how many rows (from and including row 0) to get and display.

`iloc[index]` gets and displays the row in the dataframe indicated by the integer argument `index`.

`isnull()` is a series method that checks which rows in that series have a missing value.

`fillna(value)` is a series method that returns a new series in which all missing values have been filled with the given value.

`plot()` when applied to a dataframe column of numeric values, the method displays a graph of those values. The x-axis shows the dataframe's index and the y-axis the range of the column's values. Before the method is called you first need to execute `%matplotlib inline`.

`read_csv(csvFile)` creates a dataframe from the dataset in the CSV file.

`rename(columns={oldName : newName})` renames the column `oldName` to `newName`.

`str.rstrip(suffix)` when applied to a dataframe column of string values, the method removes the argument **`suffix`** from the end of each string value in the column.

`tail()` gets and displays the last five rows of a dataframe. Optionally the method can take an integer argument to specify how many rows (until and including the last row) to get and display.

`to_datetime(aSeries)` when applied to a series, typically a column from a dataframe, this function returns a new series in which each value in **`aSeries`** has been changed to type **`datetime64`**.

Week 5: Combine and transform data Part 1

1 Life expectancy project

This week I wish to see (literally, via a chart) if the life expectancy in richer countries tends to be longer.

Figure 1

Richer countries can afford to spend more on healthcare and on road safety, for example, to reduce mortality. On the other hand, richer countries may have less healthy lifestyles.

The World Bank provides loans and grants to governments of middle and low-income countries to help reduce poverty. As part of their work, the World Bank has put together hundreds of datasets on a range of issues, such as health, education, economy, energy and the effectiveness of aid in different countries. I will use two of their datasets, which you can see online by following the links below. You do not need to download the datasets.

One dataset lists the gross domestic product (GDP) for each country, in United States dollars and cents; the other lists the life expectancy, in years, for each country. The latest life expectancy data I can access is for 2013, so that will be the year I take for the GDP. The disadvantage of using the GDP and the life expectancy values for the same year is that they do not account for the time it takes for a country's wealth to have an effect on lifestyle, healthcare and other factors influencing life expectancy.

While it is useful to have all GDPs in a common currency to compare different countries, it doesn't make much sense to report the GDP of a whole country to a supposed precision of a US cent. I noted that the value for the USA is a round number, but it is not for other countries. This is likely due in part to the conversion of local currencies to US dollars. It makes more sense to report the GDP values in a larger unit, e.g. millions of dollars. Moreover, for those who don't live in a country using the US dollar as the official currency, it's probably easier to understand GDP values in their own local currency.

To sum up, this week's project will transform currency values and combine GDP and life expectancy data.

Note that the combination is made simple by the common country names in the two datasets, but in general care has to be taken that the common attribute really means the same thing. For example, if you were combining two datasets on a common unemployment attribute, you must be sure that it was obtained in the same way as there are various ways of measuring unemployment.

I'm aware that the GDP is a crude way of comparing wealth across nations. For example, it doesn't take population or the cost of living into account. Some of this week's exercises will ask you to add the population data. Think of other ways to improve the analysis method, of other conversions that might be needed, and of other ways to investigate life expectancy factors.

Links:

GDP in current US dollars
Life expectancy at birth

1.1 Creating the data

I won't yet work with the full data. Instead I will create small tables, to better illustrate this week's concepts and techniques.

Small tables make it easier to see what is going on and to create specific data combination and transformation scenarios that test the code.

There are many ways of creating tables in pandas. One of the simplest is to define the rows as a list, with the first element of the list being the first row, the second element being the second row, etc.

Each row of a table has multiple cells, one for each column. The obvious way is to represent each row as a list too, the first element of the list being the cell in the first column, the second element corresponding to the second column, etc. To sum up, the table is represented as a list of lists.

Here is a table of the 2013 GDP of some countries, in US dollars:

In []:

```
table = [
['UK', 2678454886796.7], # 1st row
['USA', 16768100000000.0], # 2nd row
['China', 9240270452047.0], # and so on...
['Brazil', 2245673032353.8],
['South Africa', 366057913367.1]
]
```

To create a dataframe, I use a pandas function appropriately called **`DataFrame()`**. I have to give it two arguments: the names of the columns and the data itself. The column names are given as a list of strings, the first string being the first column name, etc.

In []:

```
headings = ['Country', 'GDP (US$)']
gdp = DataFrame(columns=headings, data=table)
gdp
```

Out[]:

	Country	GDP (US$)
0	UK	2.678455e+12
1	USA	1.676810e+13
2	China	9.240270e+12
3	Brazil	2.245673e+12
4	South Africa	3.660579e+11

Note that pandas shows large numbers in scientific notation, where, for example, 3e+12 means 3×10 ^{12} , i.e. a 3 followed by 12 zeros.

I define a similar table for the life expectancy, based on the 2013 World Bank data.

In []:

```
headings = ['Country name', 'Life expectancy (years)']
table = [
['China', 75],
['Russia', 71],
['United States', 79],
['India', 66],
['United Kingdom', 81]
]
life = DataFrame(columns=headings, data=table)
life
```

Out[]:

	Country name	Life expectancy (years)
0	China	75
1	Russia	71
2	United States	79

3	India	66
4	United Kingdom	81

To illustrate potential issues when combining multiple datasets, I've taken a different set of countries, with common countries in a different order. Moreover, to illustrate a non-numeric conversion, I've abbreviated country names in one table but not the other.

Exercise 1 Creating the data

Open the exercise notebook 3 and save it in the disk folder or upload it to the CoCalc project you created in Week 1. Then practise creating dataframes in Exercise 1.

If you're using Anaconda, remember that to open the notebook you'll need to navigate to it using Jupyter. Whether you're using Anaconda or CoCalc, once the notebook is open, run the existing code before you start the exercise. When you've completed the exercise, save the notebook. If you need a quick reminder of how to use Jupyter, watch again the video in Week 1 Exercise 1

1.2 Defining functions

To make the GDP values easier to read, I wish to convert US dollars to millions of US dollars.

Figure 2

I have to be precise about what I mean. For example, if the GDP is 4,567,890.1 (using commas to separate the thousands, millions, etc.), what do I want to obtain? Do I want always to round down to the nearest million, making it 4 million, round to the nearest million, making it 5, or round to one decimal place, making it 4.6 million? Since the aim is to simplify the numbers and not introduce a false sense of precision, let's round to the nearest million.

I will define my own function to do such a conversion. It's a generic function that takes any number and rounds it to the nearest million. I will later apply the function to each value in the GDP column. It's easier to first show the code and then explain it.

In []:

```
    def roundToMillions (value):
    result = round(value / 1000000)
    return result
```

A function definition always starts with **`def`** , which is a reserved word in Python.

After it comes the function's name and arguments, surrounded by parenthesis, and finally a colon (:). This function just takes one argument. If there's more than one argument, use commas to separate them.

Next comes the function's body, where the calculations are done, using the arguments like any other variables. The body must be indented, conventionally by four spaces.

For this function, the calculation is simple. I take the value, divide it by one million, and call the built-in Python function **`round()`** to convert that number to the nearest integer. If the number is exactly mid-way between two integers, **`round()`** will pick the even integer, i.e. **`round(2.5)`** is 2 but **`round(3.5)`** is 4.Finally, I write a **return statement** to pass the result back to the code that called the function. The **`return`** word is also reserved in Python.

The **`result`** variable just stores the rounded value temporarily and has no other purpose. It's better to write the body as a single line of code:

```
return round(value / 1000000)
```

Finally I need to test the function, by calling it with various argument values and checking whether the returned value is equal to what I expect.

In []:

```
roundToMillions(4567890.1) == 5
```

Out[]:

```
True
```

The art of testing is to find as few test cases as possible that cover all bases. And I mean all, especially those you think 'Naaah, it'll never happen'. It will, because data can be incorrect. Prepare for the worst and hope for the best.

So here are some more tests, even for the unlikely cases of the GDP being zero or negative, and you can probably think of others.

In []:

```
roundToMillions(0) == 0 # always test with zero...
```

Out[]:

```
True
```

In []:

```
roundToMillions(-1) == 0 #...and negative numbers
```

Out[]:

```
True
```

In []:

```
roundToMillions(1499999) == 1 # test rounding to the nearest
```

Out[]:

```
True
```

Now for the next conversion, from US dollars to a local currency, for example British pounds. I searched the internet for ‘average yearly USD to GBP rate’, chose a conversion service and took the value for 2013. Here’s the code and some tests.

In []:

```
    def usdToGbp (usd):
    return usd / 1.564768 # average rate during 2013
    usdToGbp(0) == 0
```

Out[]:

```
True
```

In []:

```
usdToGbp(1.564768) == 1
```

Out[]:

```
True
```

In []:

```
usdToGbp(-1) < 0
```

Out[]:

```
True
```

Defining functions is such an important part of coding, that you should not skip the next exercise where you will define your own functions.

Exercise 2 Defining functions

Complete Exercise 2 in the Exercise notebook 3 to practise defining your own functions.

1.3 What if...?

The third conversion, from abbreviated country names to full names, can’t be written as a simple formula, because each abbreviation is expanded differently.

Figure 3

What I need is the Python code equivalent of:

- if the name is ‘UK’, return ‘United Kingdom’,
- otherwise if the name is ‘USA’, return ‘United States’,
- otherwise return the name.

The last part basically says that if the name is none of the known abbreviations, return it unchanged. Translating the English sentence to Python is straightforward.

In []:

```
def expandCountry (name):
if name == 'UK': # if the name is 'UK'
return 'United Kingdom'
elif name == 'USA': # otherwise if the name is 'USA'
return 'United States'
else: # otherwise
return name
expandCountry('India') == 'India'
```

Out[]:

```
True
```

Note that ‘otherwise if’ is written **`'elif'`** in Python, not **`'else if'`**. As you might expect, ‘if’, ‘elif’ and ‘else’ are reserved words.

The computer will evaluate one condition at a time, from top to bottom, and execute only the instructions of the first condition that is true. Note that there is no condition after **`'else'`** , it is a ‘catch all’ in case all previous conditions fail.

Note again the colons at the end of lines and that code after the colon must be indented. That is how Python distinguishes which lines of code belong to which condition.

There are almost always many ways to write the same function. A **conditional statement** does not need to have an **'elif'** or **'else'** part. In that case, if the condition is false, nothing happens. Here is the same function, written differently.

In []:

```
def expandCountry (name):
if name == 'UK':
name = 'United Kingdom'
if name == 'USA':
name = 'United States'
return name
```

You will see later this week an example of an 'if-else' statement, i.e. without the **'elif'** part.

Exercise 3 What if…?

Complete Exercise 3 in the Exercise notebook 3 to practise writing functions with conditional statements.

1.4 Applying functions

Having coded the three data conversion functions, they can be applied to the GDP table. I first select the relevant column:

In []:

```
column = gdp['Country']
column
```

Out[]:

```
0 UK
1 USA
2 China
3 Brazil
4 South Africa
Name: Country, dtype: object
```

Next, I use the column method **apply()** , which applies a given function to each cell in the column, returning a new column, in which each cell is the conversion of the corresponding original cell:

In []:

```
column.apply(expandCountry)
```

Out[]:

```
0 United Kingdom
1 United States
2 China
```

```
3 Brazil
4 South Africa
Name: Country, dtype: object
```

Finally, I add that new column to the dataframe, using a new column heading:

In [] :

```
gdp['Country name'] = column.apply(expandCountry)
gdp
```

Out[]:

	Country	GDP (US$)	Country name
0	UK	2.678455e+12	United Kingdom
1	USA	1.676810e+13	United States
2	China	9.240270e+12	China
3	Brazil	2.245673e+12	Brazil
4	South Africa	3.660579e+11	South Africa

In a similar way, I can convert the US dollars to British pounds, then round to the nearest million, and store the result in a new column. I could apply the conversion and rounding functions in two separate statements, but using **method chaining** , I can apply both functions in a single line of code. This is possible because the column returned by the first call of **`apply()`** is the context for the second call of **`apply()`**. Here's how it's written:

In []:

```
column = gdp['GDP (US$)']
result = column.apply(usdToGbp).apply(roundToMillions)
gdp['GDP (£m)'] = result
gdp
```

Out[]:

	Country	GDP (US$)	Country name	GDP (£m)
0	UK	2.678455e+12	United Kingdom	1711727
1	USA	1.676810e+13	United States	10716029
2	China	9.240270e+12	China	5905202
3	Brazil	2.245673e+12	Brazil	1435148
4	South Africa	3.660579e+11	South Africa	233937

Now it's just a matter of selecting the two new columns, as the original ones are no longer needed.

In []:

```
headings = ['Country name', 'GDP (£m)']
gdp = gdp[headings]
gdp
```

Out[]:

	Country name	GDP (£m)
0	United Kingdom	1711727
1	United States	10716029
2	China	5905202
3	Brazil	1435148
4	South Africa	233937

Note that method chaining only works if the methods chained return the same type of value as their context, in the same way that you can chain multiple arithmetic operators (e.g. 3+4-5) because each one takes two numbers and returns a number that is used by the next operator in the chain. In this course, methods only have two possible contexts, columns and dataframes, so you can either chain column methods that return a single column (that is a **`Series`**), like **`apply()`** , or dataframe methods that return dataframes. For example, **`gdp.head(4).tail(2)`** is a dataframe just with China and Brazil, i.e. the last two of the first four rows of the dataframe shown above. You'll see further examples of chaining (and an easier way to select multiple rows) later this week.

This concludes the data transformation part. After applying functions in the next exercise, you'll learn how to combine two tables.

Exercise 4 Applying functions

You can practise applying functions in Exercise 4 of your Exercise notebook 3.

2 This week’s quiz

Check what you’ve learned this week by taking the end-of-week quiz.

Week 5 practice quiz

Open the quiz in a new window or tab then come back here when you’ve finished.

3 Summary

This week you learned how to transform currency values and combine GDP and life expectancy data by:

- creating the data
- defining the functions of data conversion
- applying the functions of data conversion.

Next week you will learn more about combining, merging and transforming tables.

Week 6: Combine and transform data Part 2

1 Joining left, right and centre

Let’s take stock for a moment. There’s the original, unchanged table (with full country names) about the life expectancy:

In []:

```
life
```

Out[]:

	Country name	Life expectancy (years)
0	China	75
1	Russia	71
2	United States	79
3	India	66
4	United Kingdom	81

… and a table with the GDP in millions of pounds and also full country names.

In []:

```
gdp
```

Out[]:

	Country name	GDP (£m)
0	United Kingdom	1711727
1	United States	10716029
2	China	5905202
3	Brazil	1435148
4	South Africa	233937

Both tables have a common column with a common name (‘Country name’). I can **join** the two tables on that common column, using the `merge()` function. Merging basically puts all columns of the two tables together, without duplicating the common column, and joins any rows that have the same value in the common column.

There are four possible ways of joining, depending on which rows I want to include in the resulting table. If I want to include only those countries appearing in the GDP table, I call the `merge()` function like so:

In []:

```
merge(gdp, life, on='Country name', how='left')
```

Out[]:

	Country name	GDP (£m)	Life expectancy (years)
0	United Kingdom	1711727	81
1	United States	10716029	79
2	China	5905202	75
3	Brazil	1435148	NaN
4	South Africa	233937	NaN

The first two arguments are the tables to be merged, with the first table being called the 'left' table and the second being the 'right' table. The `on` argument is the name of the common column, i.e. both tables must have a column with that name. The `how` argument states I want a **left join** , i.e. the resulting rows are dictated by the left (GDP) table. You can easily see that India and Russia, which appear only in the right (expectancy) table, don't show up in the result. You can also see that Brazil and South Africa, which appear only in the left table, have an undefined life expectancy. (Remember that 'NaN' stands for 'not a number.)

A **right join** will instead take the rows from the right table, and add the columns of the left table. Therefore, countries not appearing in the left table will have undefined values for the left table's columns:

In []:

```
merge(gdp, life, on='Country name', how='right')
```

Out[]:

	Country name	GDP (£m)	Life expectancy (years)
0	United Kingdom	1711727	81
1	United States	10716029	79
2	China	5905202	75
3	Russia	NaN	71
4	India	NaN	66

The third possibility is an **outer join** which takes all countries, i.e. whether they are in the left or right table. The result has all the rows of the left and right joins:

In []:

```
merge(gdp, life, on='Country name', how='outer')
```

Out[]:

	Country name	GDP (£m)	Life expectancy (years)
0	United Kingdom	1711727	81
1	United States	10716029	79
2	China	5905202	75
3	Brazil	1435148	NaN
4	South Africa	233937	NaN
5	Russia	NaN	71
6	India	NaN	66

The last possibility is an **inner join** which takes only those countries common to both tables, i.e. for which I know the GDP *and* the life expectancy. That's the join I want, to avoid any undefined values:

In []:

```
gdpVsLife = merge(gdp, life, on='Country name', how='inner')
```

Out[]:

	Country name	GDP (£m)	Life expectancy (years)
0	United Kingdom	1711727	81
1	United States	10716029	79
2	China	5905202	75

Now it's just a matter of applying the data transformation and combination techniques seen so far to the real data from the World Bank.

Exercise 5 Joining left, right and centre

Put your learning into practice by completing Exercise 5 in the Exercise notebook 3.

Remember to run the existing code in the notebook before you start the exercise. When you've completed the exercise, save the notebook.

1.1 Constant variables

You may have noticed that the same column names appear over and over in the code.

If, someday, I decide one of the new columns should be called 'GDP (million GBP)' instead of 'GDP (£m)' to make clear which currency is meant (because various countries use the pound symbol), I need to change the string in every line of code it occurs.

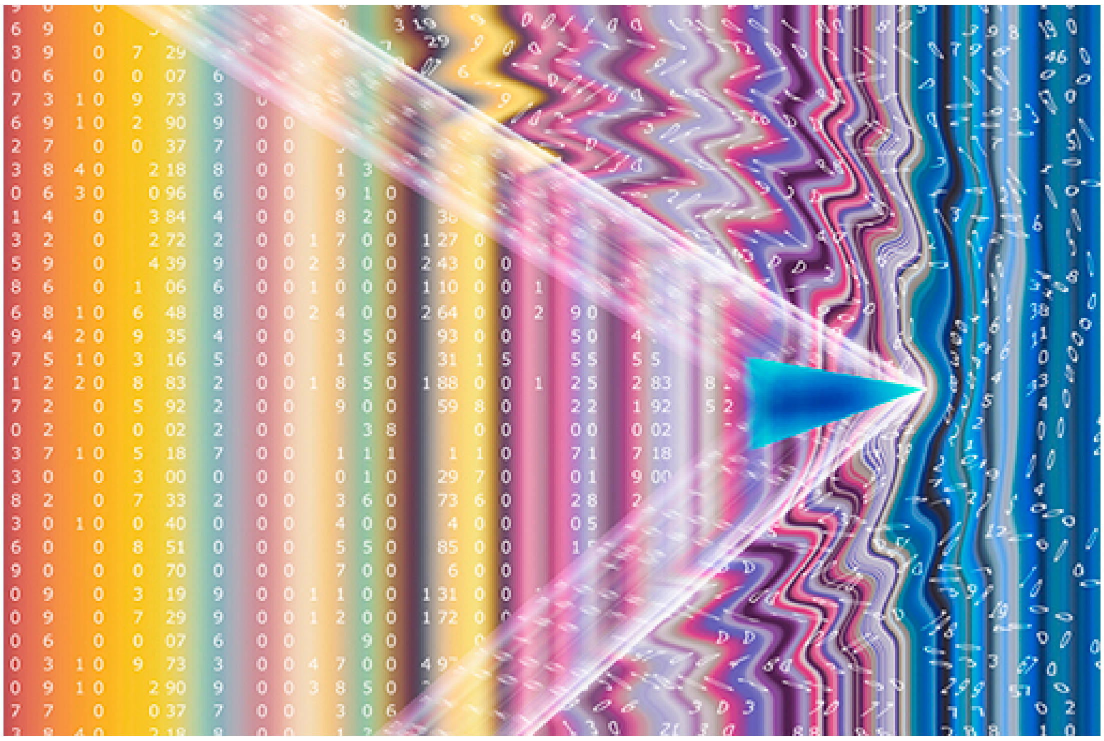


Figure 1

Laziness is the mother of invention. If I assign the string to a variable and then use the variable everywhere instead of the string, whenever I wish to change the string, I only have to edit one line of code, where it's assigned to the variable. A second advantage of using names instead of values is that I can use the name completion facility of Jupyter notebooks by pressing 'TAB'. Writing code becomes much faster…

In[]:

```
gdpInGbp = 'GDP (million GBP)'
gdpInUsd = 'GDP (US$)'
country = 'Country name'
gdp[gdpInGbp] = gdp[gdpInUsd].apply(usdToGbp)
headings = [country, gdpInGbp]
gdp = gdp[headings]
```

Such variables are meant to be assigned once. They are called **constants** , because their value never changes. However, if someone else takes my code and wishes to adapt and extend it, they may not realise those variables are supposed to remain constant. Even I may forget it and try to assign a new value further down in the code! To help prevent such slip-ups the Python convention is to write names of constants in uppercase letters, with words separated by underscores. Thus, any further assignment to a variable in uppercase will ring an alarm bell (in your head, the computer remains silent).

In[]:

```
GDP_GBP = 'GDP (million GBP)'
```

```
GDP_USD = 'GDP (US$)'
COUNTRY = 'Country name'
gdp[GDP_GBP] = gdp[GDP_USD].apply(usdToGbp)
headings = [COUNTRY, GDP_GBP]
gdp = gdp[headings]
```

Using constants is not just a matter of laziness. There are various advantages. First, constants stand out in the code.

Second, when making changes to the repeated values throughout the code, it's easy to miss an occurrence. Using constants means the code is always consistent throughout.

Third, the name of the constant can help clarify what the value means. For example, instead of using the number 1995 throughout the code, define a constant that makes clear whether it's a year, the cubic centimetres of a car engine or something else.

To sum up, using constants makes the code clearer, easier to change, and less prone to silly (but hard to find) mistakes due to inconsistent values.

Any value can be defined as a constant, whether it's a string, a number or even a dataframe. For example, you could store the data you have loaded from the file into a constant, as a reminder to not change the original data. In the rest of the week, I'll use constants mainly for the column names.

Exercise 6 Constants

To practise using constants, rewrite your exercises in the Exercise notebook 3 using them.

1.2 Getting real

Having tried out the data transformations and combination on small tables, I feel confident about using the full data from the World Bank, which I pointed you to in Life expectancy project.

Open a new browser window and go to the World Bank's data page. Type 'GDP' (without the quote marks) in the 'Find an indicator' box in the centre of the page and select 'GDP current US$'. Click 'Go'. This will take you to the data page you looked at earlier. Look at the top of your browser window. You will notice the URL is http://data.worldbank.org/indicator/NY.GDP.MKTP.CD. Every World Bank dataset is for an indicator (in this case GDP in current dollars) with a unique name (in this case NY.GDP. MKTP.CD).

Knowing the indicator name, it's a doddle to get the data directly into a dataframe, by using the **`download()`** function of the **`wb`** (World Bank) module, instead of first downloading a CSV or Excel file and then loading it into a dataframe. (Note that CoCalc's free plan doesn't allow connecting to other sites, so if you are using CoCalc you'll need to download the data as a CSV or Excel file from the World Bank and upload it to CoCalc.)

Here's the code to get the 2013 GDP values for all countries. It may take a little while for the code to fetch the data.

In []:

```
from pandas.io.wb import download
YEAR = 2013
```

```
GDP_INDICATOR = 'NY.GDP.MKTP.CD'
data = download(indicator=GDP_INDICATOR, country='all',
start=YEAR, end=YEAR)
data.head()
```

Out[]:

		NY.GDP.MKTP.CD
country	**year**	
Arab World	**2013**	2.843483e+12
Caribbean small states	**2013**	6.680344e+10
Central Europe and the Baltics	**2013**	1.418166e+12
East Asia & Pacific (all income levels)	**2013**	2.080794e+13
East Asia & Pacific (developing only)	**2013**	1.168563e+13

This table definitely has an odd shape. The three columns don't have their headings side by side, and the row numbering (0, 1, 2, etc) is missing. That's because the first two 'columns' are in fact the dataframe index. You saw a similar table in Changing a dataframe's index, when the index of the weather dataframe was set to be the 'GMT' column, with values of type **`datetime64`**. There's a dataframe method to do the inverse, i.e. to transform the row names into column values and thereby reinstate the default dataframe index.

In []:

```
gdp = data.reset_index()
gdp.head()
```

Out[]:

	country	year	NY.GDP.MKTP.CD
0	Arab World	2013	2.843483e+12
1	Caribbean small states	2013	6.680344e+10
2	Central Europe and the Baltics	2013	1.418166e+12
3	East Asia & Pacific (all income levels)	2013	2.080794e+13
4	East Asia & Pacific (developing only)	2013	1.168563e+13

I repeat the whole process for the life expectancy:

- search for 'life expectancy' on the World Bank site
- choose the 'total' dataset, which includes both female and male inhabitants
- note down its indicator (SP.DYN.LE00.IN)
- use it to get the data

- reset the dataframe index.

In []:

```
LIFE_INDICATOR = 'SP.DYN.LE00.IN'
data = download(indicator=LIFE_INDICATOR, country='all',
start=YEAR, end=YEAR)
life = data.reset_index()
life.head()
```

Out[]:

	country	year	SP.DYN.LE00.IN
0	Arab World	2013	70.086392
1	Caribbean small states	2013	71.966306
2	Central Europe and the Baltics	2013	76.127583
3	East Asia & Pacific (all income levels)	2013	74.893439
4	East Asia & Pacific (developing only)	2013	73.981255

By defining the year as a constant, it's very quick to change the code to load both datasets for any other year. If you wish to get GDP data for an earlier year than for life expectancy, then you need to define a second constant.

Exercise 7 Getting real

The approach described above requires an internet connection to download the data directly from the World Bank. That may require some time, or sometimes not even work if the connection fails. Moreover, the World Bank sometimes changes its data format, which could break the code in the rest of this week.

Therefore, the Exercise notebook 3 loads instead the GDP and life expectancy data from files WB GDP 2013.csv and WB LE 2013.csv and Exercise 7 uses the file WB POP 2013.csv , which you should add to your disk folder or CoCalc project. All files are in the normal tabular format and need no resetting of the indices.

1.3 Cleaning up

You may have noticed that the initial rows are not about countries, but groups of countries. Such aggregated values need to be removed, because we're only interested in individual countries.

The expression `frame[m:n]`, with `n` an integer bigger than `m` , represents the 'sub-table' from row `m` to row `n-1`. In other words, it is a slice of frame with exactly `n` minus `m` rows. The expression is equivalent to the more convoluted expression `frame.head(n).tail(n-m)`.

In []:

```
gdp[0:3]
```

Out[]:

	country	year	NY.GDP.MKTP.CD
0	Arab World	2013	2.843483e+12
1	Caribbean small states	2013	6.680344e+10
2	Central Europe and the Baltics	2013	1.418166e+12

To slice all rows from `m` onwards, you don't have to count how many rows there are beforehand, just omit `n`.

In []:

```
gdp[240:]
```

Out[]:

	country	year	NY.GDP.MKTP.CD
240	Uzbekistan	2013	5.679566e+10
241	Vanuatu	2013	8.017876e+08
242	Venezuela, RB	2013	3.713366e+11
243	Vietnam	2013	1.712220e+11
244	Virgin Islands (U.S.)	2013	NaN
245	West Bank and Gaza	2013	1.247600e+10
246	Yemen, Rep.	2013	3.595450e+10
247	Zambia	2013	2.682081e+10
248	Zimbabwe	2013	1.349023e+10

By trying out `head(m)` for different values of `m` , I find that the list of individual countries starts in row number 34, with Afghanistan. Hence, I slice from row 34 onwards, and that's my new dataframe.

In []:

```
    gdp = gdp[34:]
    gdp.head()
```

Out[]:

	country	year	NY.GDP.MKTP.CD
34	Afghanistan	2013	2.031088e+10
35	Albania	2013	1.291667e+10
36	Algeria	2013	2.101834e+11

37	American Samoa	2013	NaN
38	Andorra	2013	3.249101e+09

Unsurprisingly, there is missing data, so I remove those rows, as shown in Missing values in Week 4.

In []:

```
gdp = gdp.dropna()
gdp.head()
```

Out[]:

	country	year	NY.GDP.MKTP.CD
34	Afghanistan	2013	2.031088e+10
35	Albania	2013	1.291667e+10
36	Algeria	2013	2.101834e+11
38	Andorra	2013	3.249101e+09
39	Angola	2013	1.241632e+11

Finally, I drop the irrelevant year column.

In []:

```
COUNTRY = 'country'
headings = [COUNTRY, GDP_INDICATOR]
gdp = gdp[headings]
gdp.head()
```

Out[]:

	country	NY.GDP.MKTP.CD
34	Afghanistan	2.031088e+10
35	Albania	1.291667e+10
36	Algeria	2.101834e+11
38	Andorra	3.249101e+09
39	Angola	1.241632e+11

And now I repeat the whole cleaning process for the life expectancy table.

In []:

```
headings = [COUNTRY, LIFE_INDICATOR]
life = life[34:].dropna()[headings]
```

```
life.head()
```

Out[]:

	country	SP.DYN.LE00.IN
34	Afghanistan	60.931415
35	Albania	77.537244
36	Algeria	71.009659
39	Angola	51.866171
40	Antigua and Barbuda	75.829293

Note how a single line of code can chain a row slice, a method call and a column slice, because each takes a dataframe and returns a dataframe.

Exercise 8 Cleaning up

Clean up the population data from Exercise 7, in Exercise 8 in the exercise notebook 3.

1.4 Joining and transforming

With the little tables, I first transformed the columns and then joined the tables.

Figure 2

As you may be starting to realise, there’s often more than one way to do it. Just for illustration, I’ll do the other way round for the big tables. Here are the tables, as a reminder.

In []:

```
life.head()
```

Out[]:

	country	SP.DYN.LE00.IN
34	Afghanistan	60.931415
35	Albania	77.537244
36	Algeria	71.009659
39	Angola	51.866171
40	Antigua and Barbuda	75.829293

In []:

```
gdp.head()
```

Out[]:

	country	NY.GDP.MKTP.CD
34	Afghanistan	2.031088e+10
35	Albania	1.291667e+10

36	Algeria	2.101834e+11
38	Andorra	3.249101e+09
39	Angola	1.241632e+11

First, an inner join on the common column to combine rows where the common column value appears in both tables.

In []:

```
gdpVsLife = merge(gdp, life, on='country', how='inner')
gdpVsLife.head()
```

Out []:

	country	NY.GDP.MKTP.CD	SP.DYN.LE00.IN
0	Afghanistan	2.031088e+10	60.931415
1	Albania	1.291667e+10	77.537244
2	Algeria	2.101834e+11	71.009659
3	Angola	1.241632e+11	51.866171
4	Antigua and Barbuda	1.200588e+09	75.829293

Second, the dollars are converted to millions of pounds.

In []:

```
GDP = 'GDP (£m)'
column = gdpVsLife[GDP_INDICATOR]
gdpVsLife[GDP] = column.apply(usdToGbp).apply(roundToMillions)
gdpVsLife.head()
```

Out[]:

	country	NY.GDP.MKTP.CD	SP.DYN.LE00.IN	GDP (£m)
0	Afghanistan	2.031088e+10	60.931415	12980
1	Albania	1.291667e+10	77.537244	8255
2	Algeria	2.101834e+11	71.009659	134322
3	Angola	1.241632e+11	51.866171	79349
4	Antigua and Barbuda	1.200588e+09	75.829293	767

Third, the life expectancy is rounded to the nearest integer, with a by now familiar function.

In []:

```
LIFE = 'Life expectancy (years)'
```

```
gdpVsLife[LIFE] = gdpVsLife[LIFE_INDICATOR].apply(round)
gdpVsLife.head()
```

Out[]:

	country	NY.GDP. MKTP.CD	SP.DYN. LE00.IN	GDP (£m)	Life expectancy (years)
0	Afghanistan	2.031088e+10	60.931415	12980	61
1	Albania	1.291667e+10	77.537244	8255	78
2	Algeria	2.101834e+11	71.009659	134322	71
3	Angola	1.241632e+11	51.866171	79349	52
4	Antigua and Barbuda	1.200588e+09	75.829293	767	76

Lastly, the original columns are discarded.

In []:

```
headings = [COUNTRY, GDP, LIFE]
gdpVsLife = gdpVsLife[headings]
gdpVsLife.head()
```

Out[]:

	country	GDP (£m)	Life expectancy (years)
0	Afghanistan	12980	61
1	Albania	8255	78
2	Algeria	134322	71
3	Angola	79349	52
4	Antigua and Barbuda	767	76

For the first five countries there doesn’t seem to be any relation between wealth and life expectancy, but that might be just for those countries.

Exercise 9 Joining and transforming

Have a go at merging dataframes with an inner join in Exercise 9 in the Exercise notebook 3.

2 Correlation

To see if life expectancy grows when the GDP increases I will use a statistical measure known as the **Spearman rank correlation coefficient**.

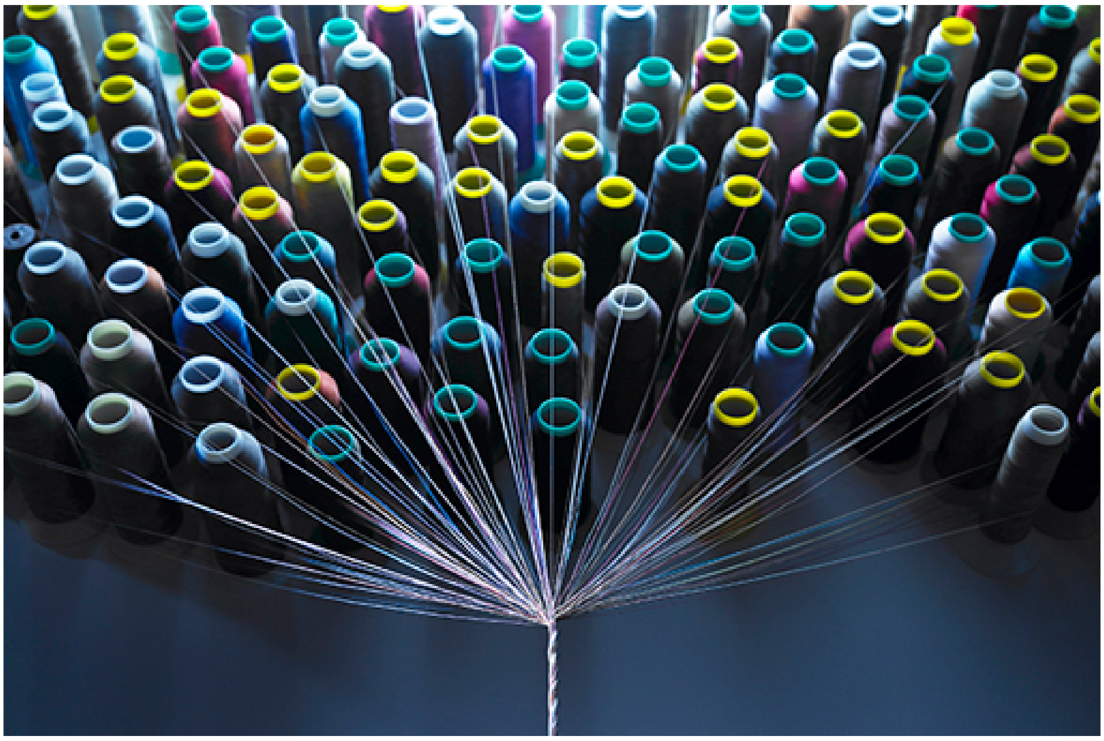


Figure 3

It's a number between -1 and 1 that describes how well two indicators correlate, in the following sense.

- A value of 1 means that if I rank (sort) the data from smallest to largest value in one indicator, it will also be in ascending order according to the other indicator. In other words, if one indicator grows, so does the other.
- A value of -1 means a perfect inverse rank relation: if I sort the data from smallest to largest according to one indicator, I will see it is sorted from largest to smallest in the other indicator. When one indicator goes up, the other goes down.
- A value of 0 means there is no rank relation between the two indicators.

A positive value smaller than 1 (or a negative value larger than -1) means there is some direct (or inverse) correlation, but it is not systematic across the whole dataset.

The **p-value** indicates how significant the result is, in a particular technical sense. To say a correlation is statistically significant doesn't necessarily mean it is important or strong in the real world, but only that there is reasonable statistical evidence that there is some kind of relationship. Typically, the obtained correlation coefficient is considered statistically significant if the p-value is below 0.05.

The pandas module doesn't calculate complex statistics. There are other modules in the Anaconda distribution for that. In particular, `scipy` (Scientific Python) has a stats module that provides the `spearmanr()` function. The function takes as arguments the two columns of data to correlate. Contrary to the functions you've seen so far, it returns two values instead of one: the correlation and the p-value. To store both values, simply use a pair of variables, written in parenthesis.

To show the results in a nicer way, I will use the Python **`print()`** function, which displays its arguments in a single line.

In []:

```
from scipy.stats import spearmanr
gdpColumn = gdpVsLife[GDP]
lifeColumn = gdpVsLife[LIFE]
(correlation, pValue) = spearmanr(gdpColumn, lifeColumn)
print('The correlation is', correlation)
if pValue < 0.05:
print('It is statistically significant.')
else:
print('It is not statistically significant.')
```

Out[]:

```
The correlation is 0.493179132478.
It is statistically significant.
```

Although there is a statistically significant direct correlation (life expectancy grows as GDP grows), it isn’t strong.

A perfect (direct or inverse) correlation doesn’t mean there is any cause-effect between the two indicators. A perfect direct correlation between life expectancy and GDP would only state that the higher the GDP, the higher the life expectancy. It would not state that the higher expectancy is due to the GDP. Correlation is not causation.

Exercise 10 Correlation

Calculate the correlation between GDP and population in Exercise 10 in the Exercise notebook 3.

Remember to run the existing code in the notebook before you start the exercise. When you’ve completed the exercise, save the notebook.

2.1 Scatterplots

Statistics can be misleading. A coefficient of zero only states there is no ranking relation between the indicators, but there might be some other relationship.

In the next example, the correlation between x and y is zero, but they are clearly related (y is the square of x).

In []:

```
table = [ [-2,4], [-1,1], [0,0], [1,1], [2,4] ]
data = DataFrame(columns=['x', 'y'], data=table)
(correlation, pValue) = spearmanr(data['x'], data['y'])
print('The correlation is', correlation)
data
```

Out[]:

```
The correlation is 0.0
```

	x	y
0	-2	4
1	-1	1
2	0	0
3	1	1
4	2	4

It’s therefore best to complement the quantitative analysis with a more qualitative view provided by a chart. In the case of correlations, **scatterplots** will do very nicely. Each country is a dot plotted at the x and y coordinates corresponding to the GDP and life expectancy values.

In []:

```
%matplotlib inline
gdpVsLife.plot(x=GDP, y=LIFE, kind='scatter', grid=True)
```

Out[]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x10e2e6eb8>
```

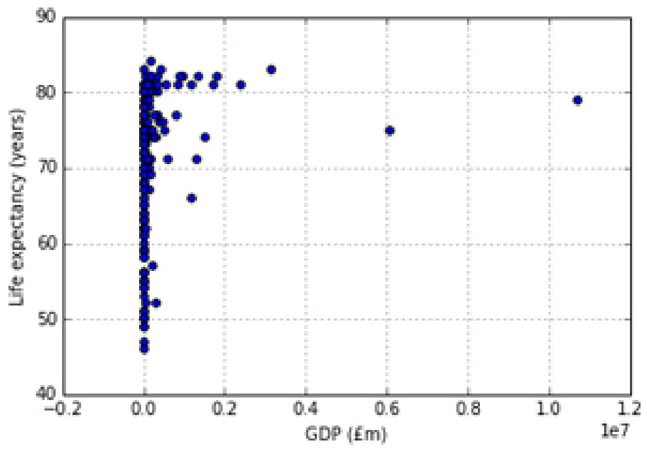


Figure 4

This graph is not very useful. The GDP difference between the poorest and richest countries is so vast that the whole chart is squashed to fit all GDP values on the x-axis. It is best to use a **logarithmic scale** , where the axis values don’t increase by a constant interval (10, 20, 30, for example), but by a multiplicative factor (10, 100, 1000, 10000, etc.). The parameter `logx` has to be set to `True` to get a logarithmic scale on the x-axis. Moreover, let’s make the chart a bit wider, by using the `figsize` parameter you saw last week.

In []:

```
gdpVsLife.plot(x=GDP, y=LIFE, kind='scatter', grid=True,
logx=True, figsize = (10, 4))
```

Out[]:

```
>matplotlib.axes._subplots.AxesSubplot at 0x10e400588>
```

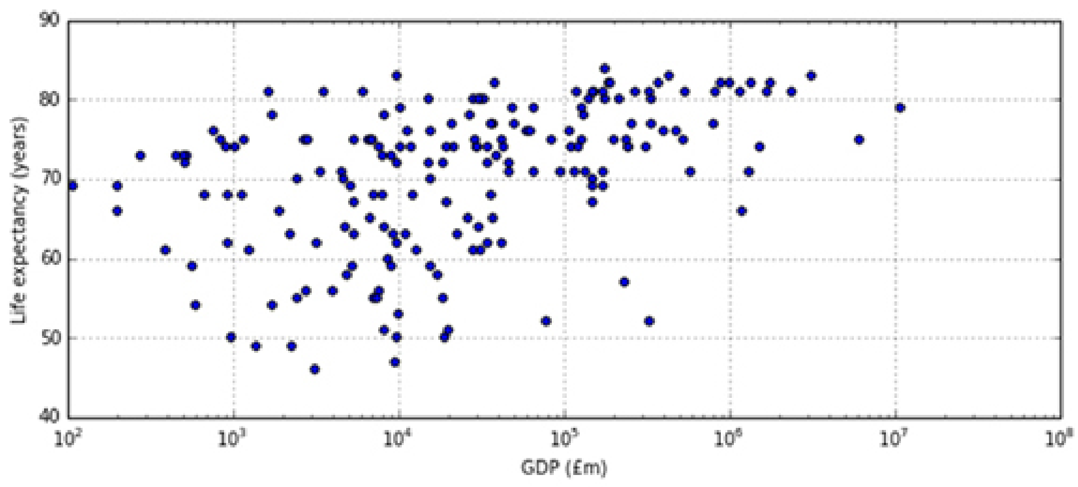


Figure 5

The major tick marks in the x-axis go from 10 ^{2} (that's a one followed by two zeros, hence 100) to 10 ^{8} (that's a one followed by eight zeros, hence 100,000,000) million pounds, with the minor ticks marking the numbers in between. For example, the eight minor ticks between 10 ^{2} and 10 ^{3} represent the values 200 (2 × 10 ^{2}), 300 (3 × 10 ^{2}), and so on until 900 (9 × 10 ^{2}). As a further example, the country with the lowest life expectancy is on the second minor tick to the right of 10 ^{3} , which means its GDP is about 3 × 10 ^{3} (three thousand) million pounds.

Countries with a GDP around 10 thousand (10 ^{4}) millions of pounds have a wide range of life expectancies, from under 50 to over 80, but the range tends to shrink both for poorer and for richer countries. Countries with the lowest life expectancy are neither the poorest nor the richest, but those with highest expectancy are among the richer countries.

Exercise 11 Scatterplots

Practise using Scatterplots in Exercise11 in the Exercise notebook 3.

2.2 My project

I've written up my analysis of this week's project in the notebook you can open this in your downloaded files.

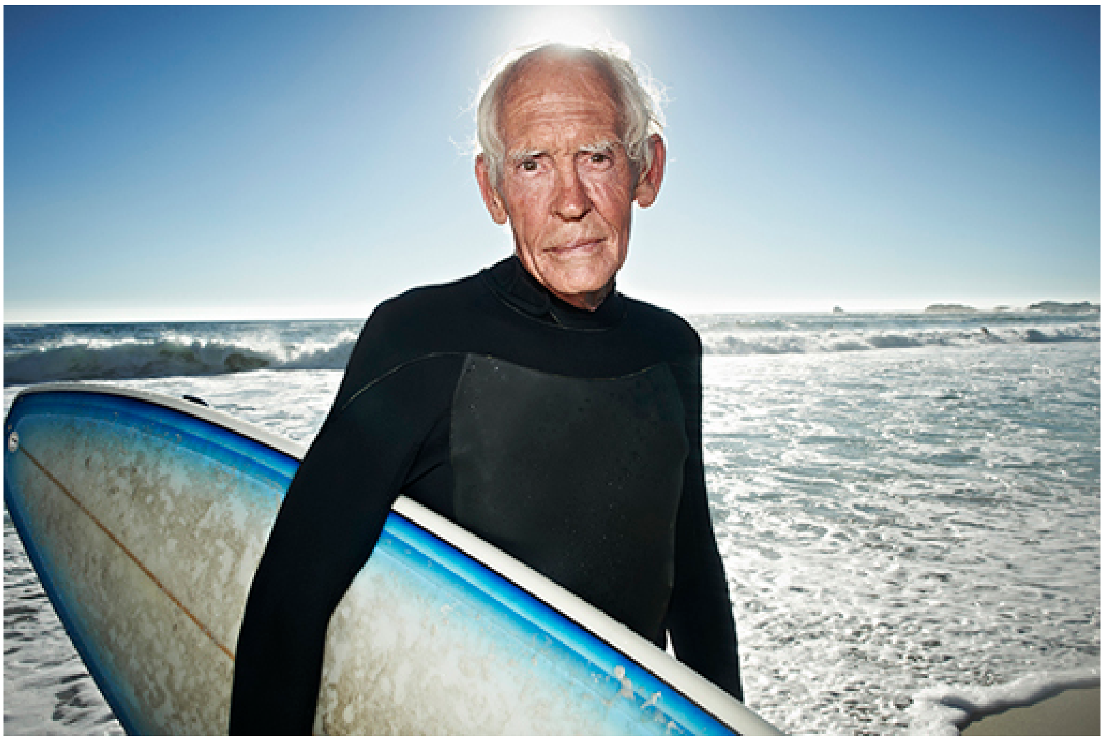


Figure 6

The structure is very simple: besides the introduction and the conclusions, there is one section for each step of the analysis – downloading, cleaning, transforming, and merging the data, then calculating and visualising the correlation.

Open Project 4: Life expectancy

If you have time, extend my project to answer different questions or create your own project in the activity below.

Activity 1

Extend the project

Make a copy of the Project 3: GDP and Life expectancy and change it to answer one or more of the following questions:

- To what extent do the ten countries with the highest GDP coincide with the ten countries with the longest life expectancy?
- Which are the two countries in the right half of the plot (higher GDP) with life expectancy below 60 years? What factors could explain their lower life expectancy compared to countries with similar GDP?

  Hint: use the filtering techniques you learned in Week 2 to find the two countries.
- Redo the analysis using the countries' GDP per capita (i.e. per inhabitant) instead of their total GDP. If you've done the workbook exercises, you already have a column with the population data.

  Hint: write an expression involving the GDP and population columns, as you learned in Calculating over columns in Week 1.
- Think about the units in which you display GDP per capita.

- Redo the analysis using the indicator suggested at the end of the project notebook.

Create your own project

If you have more time, create a completely new project and choose another two of the hundreds of World Bank indicators and see if there is any correlation between them. If there is a choice of similar indicators, choose one that leads to meaningful comparisons between countries.

Look at the results you obtained and take a few moments to assess how they differ from mine.

3 This week’s quiz

Check what you’ve learned this week by taking the end-of-week quiz.

Week 6 practice quiz

Open the quiz in a new window or tab then come back here when you’ve finished.

4 Summary

Figure 7

This week you transformed and combined databy:

- computing a correlation coefficient between two series of values and checking whether the correlation is statistically significant
- generating scatterplots to ook for other relationships
- using a logarithmic scale when an indicator had a wide range of values.

Next week you'll learn how to group, export and import data to generate pivot table style reports.

4.1 Weeks 5 and 6 glossary

Here are alphabetical lists, for quick look up, of what this week introduced.

Concepts

A **conditional statement** is of the form

```
if condition1:
statements1
elif condition2:
statements2
...
else:
statements
```

The computer evaluates the conditions from top to bottom and executes *only* the statements for the *first* condition that is true. If all conditions are false, it executes the **`else`** statements. If there is no **`else`** part nothing happens. The **`elif`** parts are optional too. Each block of statements must be indented, usually by four spaces.

A **constant** is a variable that is assigned only once, i.e. its initial value never changes. Constant names are conventionally written in uppercase, with underscores to separate multiple words.

A **function definition** is typically of the form

```
def functionName (argumentName1, argumentName2,...):
statements using arguments to compute the result
return result
```

All statements in the body of the function must have the same indentation, usually four spaces. The statements use the arguments like normal variables. The execution of the function ends when a return statement is encountered.

A **join** is the merging of two tables on a common column. The resulting table has all columns of both tables (the common column isn't duplicated), and the rows are determined by the type of join. Rows in the two tables that have the same value in the common column become a joined row in the resulting table.

In a **logarithmic scale** , each major tick represents a value that is the multiplication by some constant (usually 10) of the value of the previous major tick.

A **method chain** is an expression like **`context.method1(args1).method2(args2).method3(args3)`** where each method has and returns the same type of context, except possibly the last method, which can return any type of value.

The **p-value** is an indication of the significance of the result. Usually a p-value below 0.05 is taken to mean the result is statistically significant.

A **return statement** is of the form **`return expression`** and passes the value of the expression back to the code that called the function to which the return statement belongs.

The **Spearman rank correlation coefficient** of two series of values (e.g. two columns) is a number from -1 (perfect inverse correlation) to 1 (perfect direct correlation), with 0 meaning there is no rank correlation. Correlation doesn't imply causation. A rank correlation of 1 merely states that both values increase and decrease together, while a correlation of -1 states that if one value increases, the other decreases.

A **test** is some code that checks whether some other code works as expected, e.g. a boolean expression that compares the return value of a function call with the expected value.

Reserved Words

`def, elif, else,` if and **`return`** cannot be used as names.

Functions and methods

`col.apply(functionName)` returns a new column, obtained by applying the given one-argument function to each cell in column **`col`**.

`DataFrame(columns=listOfStrings, data=listOfLists)` returns a new dataframe, given the data as a list of rows, each row being a list of values in column order.

`download(indicator=string, country='all', start=number, end=number)` is a function in the pandas.io.wb module that downloads the World Bank data for the given indicator and all countries and country groups from the given start year to the given end year.

`merge(left=frame1, right=frame2, on=columnName, how=string)` returns a new dataframe, obtained by joining the two frames on the columns with the given common name. The **`how`** argument can be one of **`‘left’, ‘right’, ‘inner’`** and **`'outer’.`**

`print()` is a Python function that takes one or more expressions and prints their values on the screen in a single line.

`frame.reset_index()` returns a new dataframe in which rows are labelled from 0 onwards.

`spearmanr()` is a function in the scipy.stats module that takes two columns and returns a pair of numbers: the Spearman rank correlation coefficient of the two series of values, and its p-value.

Week 7: Further techniques Part 1

1 I spy with my little eye

One of the ways you are shown for loading World Bank data into the notebook in Week 7, was to use the **`download ()`** function.

Figure 1

One way to find out for yourself what sorts of argument a function expects is to ask it. Running a code cell containing a question mark (?) followed by a function name should pop up a help area in the bottom of the notebook window. (Close it using the x in the top right hand corner of the panel.)

In []:

```
from pandas.io.wb import download
?download
```

The function documentation tells you that you can enter a list of one or more country names using standard country codes as well as a date range. You can also calculate a date range from a single date to show the **`N`** years of data leading up to a particular year.

Note that if you are using the CoCalc free plan, you will not be able to use the **`download ()`** function to download the data directly from the World Bank API, although you will still be able to inspect the documentation associated with the function.

In []:

```
YEAR = 2013
GDP_INDICATOR = 'NY.GDP.MKTP.CD'
gdp = download(indicator=GDP_INDICATOR, country=['GB','CN'], start=YEAR-5, end=YEAR)
gdp = gdp.reset_index()
gdp
```

Out[]:

	country	year	NY.GDP.MKTP.CD
0	China	2013	9.490603e+12
1	China	2012	8.461623e+12
2	China	2011	7.492432e+12
3	China	2010	6.039659e+12
4	China	2009	5.059420e+12
5	China	2008	4.558431e+12
6	United Kingdom	2013	2.678173e+12
7	United Kingdom	2012	2.614946e+12
8	United Kingdom	2011	2.592016e+12
9	United Kingdom	2010	2.407857e+12
10	United Kingdom	2009	2.308995e+12
11	United Kingdom	2008	2.791682e+12

Although many datasets that you are likely to work with are published in the form of a single data table, such as a single CSV file or spreadsheet worksheet, it is often possible to regard the dataset as being made up from several distinct subsets of data.

In the above example, you will probably notice that each country name appears in several rows, as does each year. This suggests that we can make different sorts of comparisons between different groupings of data using just this dataset. For example, compare the total GDP of each country calculated over the six years 2008 to 2013 using just a single line of code:

In []:

```
gdp.groupby('country')['NY.GDP.MKTP.CD'].aggregate(sum)
```

Out[]:

```
country
China 4.110217e+13
United Kingdom 1.539367e+13
```

```
Name: NY.GDP.MKTP.CD, dtype: float64
```

Essentially what this does is to say ‘for each country, find the total GDP’.

The total combined GDP for those two countries in each year could be found by making just one slight tweak to our code (can you see below where I made the change?):

In []:

```
gdp.groupby('year')['NY.GDP.MKTP.CD'].aggregate(sum)
```

Out[]:

```
year
2008 7.350113e+12
2009 7.368415e+12
2010 8.447515e+12
2011 1.008445e+13
2012 1.107657e+13
2013 1.216878e+13
Name: NY.GDP.MKTP.CD, dtype: float64
```

That second calculation probably doesn’t make much sense in this particular case, but what if there was another column saying which region of the world each country was in? Then, by taking the data for all the countries in the world, the total GDP could be found for each region by grouping on *both* the year *and* the region.

Next, you will consider ways of grouping data.

1.1 Ways of grouping data

Think back to the weather dataset you used in Week 3 , how might you group that data into several distinct groups? What sorts of comparisons could you make by grouping just the elements of that dataset? Or how might you group and compare the GDP data?

Figure 2

One thing the newspapers love to report are weather ‘records’, such as the ‘hottest June ever’ or the wettest location in a particular year as measured by total annual rainfall, or highest average monthly rainfall. How easy is it to find that information out from the data?

Or with the GDP data, if countries were assigned to economic groupings such as the European Union, or regional groupings such as Africa, or South America, how would you generate information such as lowest GDP in the EU or highest GDP in South America?

This week you will learn how to split data into groups based on particular features of the data, and then generate information about each separate group, across all of the groups, at the same time.

Activity 1 Grouping data

Based on the data you have seen so far, or some other datasets you may be aware of, what other ways of grouping data can you think of, and why might grouping data that way be useful?

Provide your answer...

1.2 Data that describes the world of trade

A news article from the *Guardian* announcing a gloomy export outlook for UK manufacturers (see the link below), got me wondering about what sorts of thing different countries actually export.

For example, it might surprise you that India was the world's largest exporter by value of unset diamonds in 2014 (24 billion US dollars worth), or that Germany was the biggest importer of chocolate (over $2.5 billion worth) in that same year.

National governments all tend to publish their own trade figures, but the UN also collect data from across the world. In particular, the UN's global trade database, Comtrade, contains data about import and export trade flows between countries for a wide range of goods and services.

Figure 3

So if you've ever wondered where your country imports most of its T-shirts from, or exports most of its municipal waste to, Comtrade is likely to have the data.

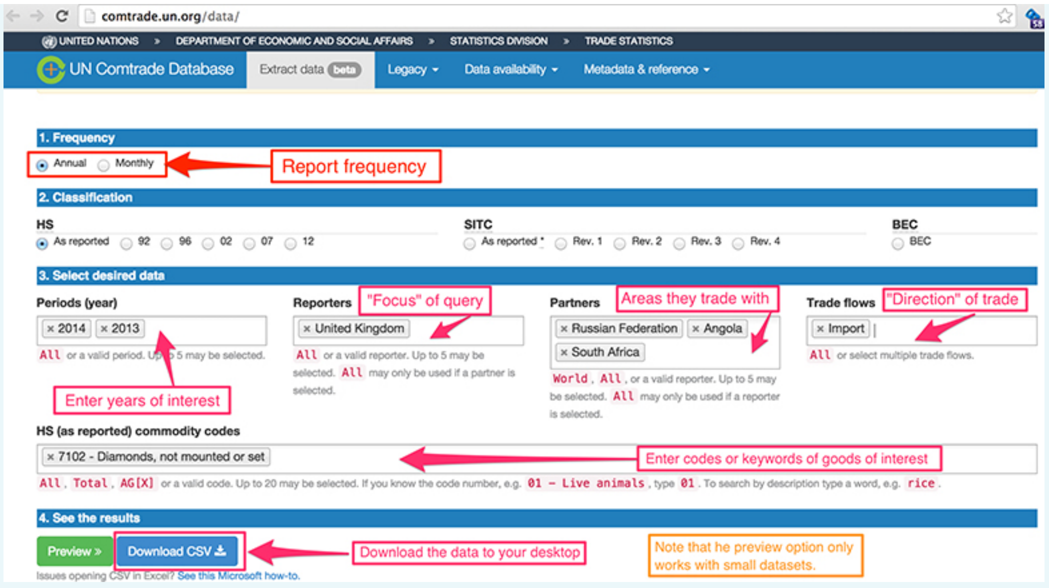
In the next section, you will find out about the Comtrade data.

1.3 Exploring the world of export data

The Comtrade Data Extraction interface provides a user interface for selecting, previewing and exporting data from the Comtrade database.

Activity 2 Exploring export data

Open the Comtrade Data Extraction interface and keep it open alongside this page. You'll explore the options and preview some data.


comtrade.un.org/data/
UNITED NATIONS » DEPARTMENT OF ECONOMIC AND SOCIAL AFFAIRS » STATISTICS DIVISION » TRADE STATISTICS
UN Comtrade Database
Extract data beta
Legacy
Data availability
Metadata & reference
Report frequency
"Focus" of query
Areas they trade with
"Direction" of trade
Enter years of interest
Enter codes or keywords of goods of interest
Preview »
Download CSV
Download the data to your desktop
Note that he preview option only works with small datasets.

1. Frequency

[✓ Annual] Monthly

2. Classification

HS: [✓ As reported] 92 96 02 07 12

SITC: As reported * Rev. 1 Rev. 2 Rev. 3 Rev. 4

BEC: BEC

3. Select desired data

Periods (year): × 2014 × 2013
All or a valid period. Up to 5 may be selected.

Reporters: × United Kingdom
All or a valid reporter. Up to 5 may be selected. All may only be used if a partner is selected.

Partners: × Russian Federation × Angola × South Africa
World, All, or a valid reporter. Up to 5 may be selected. All may only be used if a reporter is selected.

Trade flows: × Import
All or select multiple trade flows.

HS (as reported) commodity codes: × 7102 - Diamonds, not mounted or set
All, Total, AG[X] or a valid code. Up to 20 may be selected. If you know the code number, e.g. 01 - Live animals, type 01. To search by description type a word, e.g. rice.

4. See the results

Issues opening CSV in Excel? See this Microsoft how-to.

Figure 4: Comtrade Data Extraction interface

In the text area marked **HS (as reported) commodity codes** , start to enter the name of various goods and services. You should see suggestions regarding different goods and services that Comtrade records trade flow data for.

If you don't select too much data, you should be able to get a preview of the data by clicking the green 'Preview' button. Notice that the interface allows you to sort the data by a particular column, which provides a quick way of finding the countries that export most, or least, goods by value.

If you selected 'All' reporters, you will probably notice that a decreasing sort on the 'Trade Value' column always has 'World' at the top: in the 'All' reports dataset, individual country reports and reports from 'areas not elsewhere specified' ('nes') are complemented by the 'World' report which represents a sum total of those other values.

The user interface is rather complicated at first glance, but with a bit of trial and error you should be able to work out:

- how to display trade flows between a particular country (the 'Reporter') and a particular country or region of the world (the 'Partners')
- how to limit the display to show just imports, or exports, between 'Reporter(s)' and 'Partner(s)'
- how to display data for different years
- how to display data for different months in a particular year, or all the months in a particular year.

You might notice that the commodities codes are organised hierarchically, i.e. a code breaks down into further sub-codes. For example:

- 3825 – Residual products of the chemical or allied industries
  - 382510 – Municipal waste
  - 382520 – Sewage sludge
  - 382530 – Clinical waste
  - …

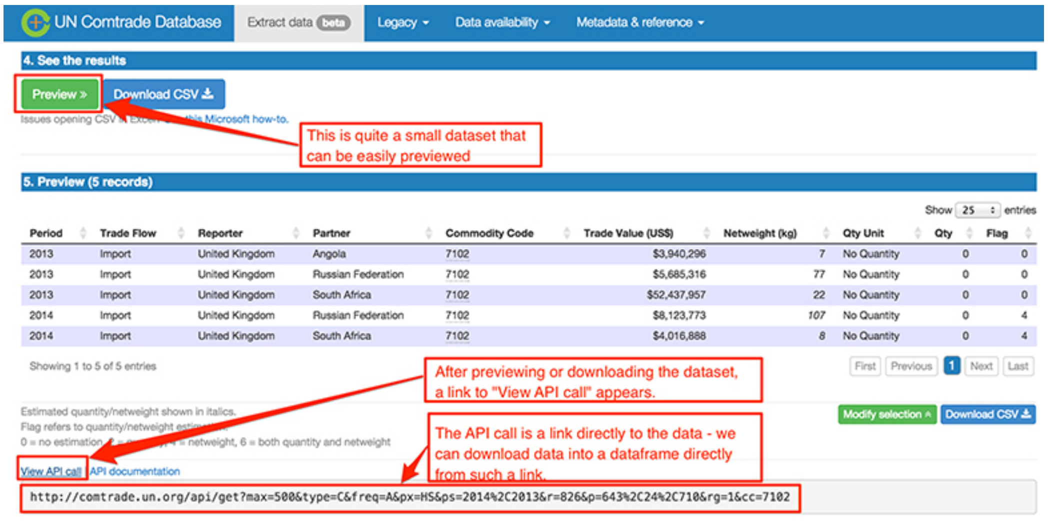
> Adding up the results from the next level down on a particular code should generate trade value totals that correspond to the higher level totals, rounding errors aside. This means that if you want to focus on the subcategories of a particular commodity type, you may well be able to do so.
>
> For a particular category of goods, and a reporting period of a single month or year, select your country as the reporter and ‘All’ as the partner.
>
> Does the range of goods and services listed within the database surprise you?

Keep the Comtrade webpage open as you’ll use it again in the next section.

1.4 Getting data from the Comtrade API

Hopefully, you have a few ideas about data you’d like to explore from the Comtrade database.

In the previous section, I managed to identify a set of data that describes the amount of unset diamonds (commodity code 7102) imported into the UK from the Russian Federation, Angola and South Africa in 2013 and 2014.



Period	Trade Flow	Reporter	Partner	Commodity Code	Trade Value (US$)	Netweight (kg)	Qty Unit	Qty	Flag
2013	Import	United Kingdom	Angola	7102	$3,940,296	7	No Quantity	0	0
2013	Import	United Kingdom	Russian Federation	7102	$5,685,316	77	No Quantity	0	0
2013	Import	United Kingdom	South Africa	7102	$52,437,957	22	No Quantity	0	0
2014	Import	United Kingdom	Russian Federation	7102	$8,123,773	*107*	No Quantity	0	4
2014	Import	United Kingdom	South Africa	7102	$4,016,888	*8*	No Quantity	0	4

Showing 1 to 5 of 5 entries
First
Previous
1
Next
Last
After previewing or downloading the dataset, a link to "View API call" appears.
Modify selection
Download CSV
Estimated quantity/netweight shown in italics.
The API call is a link directly to the data - we can download data into a dataframe directly from such a link.
View API call
API documentation

```
http://comtrade.un.org/api/get?max=500&type=C&freq=A&px=HS&ps=2014%2C2013&r=826&p=643%2C24%2C710&rg=1&cc=7102
```

Figure 5 Comtrade Data Extraction interface

You can export the data you have selected as a CSV file that will be downloaded to your own computer by clicking on the *Download CSV* button. You may find it useful to change the filename of the downloaded file to something more meaningful than the comtrade.csv default name.

If you moved the downloaded CSV file into the same folder as your Exercise notebook 4 (that you’ll download later), you could use the following command to load the data into a pandas dataframe:

In []:

```
    filename='comtrade.csv
    ' df=read_csv(filename, dtype={'Commodity Code':str, 'Reporter Code':str })
```

The ‘Commodity Code’ and ‘Reporter Code’ values are explicitly read in as a string **(str)** otherwise codes like 0401 will be returned as the number 401.

One of the problems of working with real data like this is that it may not be just the data you want. The data returned from Comtrade includes several columns that are essentially surplus to requirements for the reports you will produce. I suggest that you clean the dataframes so that they contain at most the following key columns: ‘Year’, ‘Period’, ‘Trade Flow’, ‘Reporter’, ‘Partner’, ‘Commodity’, ‘Commodity Code’, ‘Trade Value (US$)’.

In []:

```
COLUMNS = ['Year', 'Period','Trade Flow','Reporter','Partner', 'Commodity','Commodity
Code','Trade Value (US$)']
df=df[COLUMNS]
```

To avoid conflating data relating to all countries (the ‘World’ partner), and each separate country, create separate dataframes for each, using the comparison operators introduced in Week 3.

In []:

```
world = df[df['Partner'] == 'World']
countries = df[df['Partner'] != 'World']
```

A More Direct Way of Getting the Data

Just as there was a method for downloading data directly from the World Bank, there is also a more direct way of getting the Comtrade data into a dataframe – directly from the Comtrade website. You might have noticed that when you downloaded the file from the Comtrade website, a link appeared on the site labelled ‘View API Call’.

An API is an ‘application programming interface’ that provides a means for one computer to talk to another ‘in machine terms’. When you extracted data from the World Bank, you were calling the World Bank API using a set of functions provided by the pandas library. Behind the scenes, these functions create URLs (that is, web addresses) that call the World Bank API and allow requests to be made directly from it, putting the response into a pandas dataframe.

In the case of Comtrade, clicking the *View API Link* reveals a URL that requests the data you selected in the search form as a data file, though not, by default, as a CSV data file.

This link can be used to download data directly into a pandas dataframe from Comtrade, although you will need to make a couple of modifications to the URL first. In particular, change the max value to 5000 (to increase the amount of data returned by each request) and add `&fmt=csv` to the end of the URL to ensure that the data is returned in a CSV format.

For example, if you copied the URL:

http://comtrade.un.org/api/get?max=500&type=C&freq=M&px=HS&ps=2015&r=826&-p=all&rg=1%2C2&cc=0401%2C0402

you would need to modify it as follows:

http://comtrade.un.org/api/get?max= **5000** &type=C&freq=M&px=HS&ps=2015&r=826&-p=all&rg=1%2C2&cc=0401%2C0402 **&fmt=csv**

You can then load the data in using the panda `read_csv()` function.

Note that if you are using the CoCalc free plan, you will not be able to download data directly from the Comtrade API into a pandas dataframe.

Set the datatypes as shown using the `dtype` argument to ensure that the codes are read in correctly.

In []:

```
URL='http://comtrade.un.org/api/get?max=5000&type=C&freq=A&px=HS&ps=2014%2C2013%
2C2012&r=826&p=all&rg=all&cc
=0401%2C0402&fmt=csv'
df=read_csv(URL, dtype={'Commodity Code':str, 'Reporter Code':str})
```

Having downloaded the data, you should then separate out the World data as before.

If you want to save a copy of data downloaded into pandas directly from the Comtrade API, call the **`to_csv()`** method on your dataframe, pasting in the filename you want to save the file under, and setting **`index=False`** so that the dataframe's automatically introduced index column is not included. For example:

```
countries.to_csv('saved_country_data_example.csv', index=False)
```

The file will be saved in the same folder as the notebook.

1.5 Practice getting data

Exercise 1 Getting data from API

In Exercise 1, identify a dataset from the Comtrade Data Extraction interface selecting one or more commodity codes and a single reporter that are of interest to you and import the data into pandas.

Open the exercise notebook 4 and save it in the disk folder or the CoCalc project you created in Week 1. You should also open the comtrade_pivot.html comtrade_milk_uk_monthly_14.csv files and save them into the same folder or project.

Remember to run the existing code in the notebook before you start the exercise. When you've completed the exercise, save the notebook. If you need a quick reminder of how to use Jupyter watch again the video in Week 1 Exercise 1.

For the commodities and reporter you chose, find out which countries are the biggest partners in recent years in terms of import and export trade flows.

2 This week's quiz

Check what you've learned this week by taking the end-of-week quiz.

Week 7 practice quiz

Open the quiz in a new window or tab then come back here when you've finished.

3 Summary

This week you’ve learned how to take a dataset that contains multiple possible groupings, or subsets of data, and work with those groups to perform a variety of transformations. You’ve explored:

- ways of grouping data
- Comtrade data
- the world of export data
- how to get data.

Next week looks at how to split the data contained in a dataframe into multiple groups based on the unique ‘key’ values in a single column, or unique combinations of values that appear across two or more columns.

Week 8: Further techniques Part 2

1 The split-apply-combine pattern

In the exercise in Week 7, you downloaded data from Comtrade that could be described as ‘heterogenous’ or mixed in some way. For example, the same dataset contained information relating to both imports and exports.

To find the partner countries with the largest trade value in terms of exports means filtering the dataset to obtain just the rows containing export data and then ranking those. Finding the largest import partner requires a sort on just the import data.

Figure 1

But what if you wanted to find out even more refined information? For example:

- the total value of exports of product X from the UK to those countries on a year by year basis (group the information by year and then find the total for each year)
- the total value of exports of product X from the UK to each of the partner countries by year (group the information by country and year and then find the total for each country/year pairing)

- the average value of exports across all the countries on a month by month basis (group by month, then find the average value per month)
- the average value of exports across each country on a month by month basis (group by month and country, then find the average value over each country/month pairing)
- the difference month on month between the value of imports from, or exports to, each particular country over the five year period (group by country, order by month and year, then find the difference between consecutive months).

In each case, the original dataset needs to be separated into several subsets, or groups of data rows, and then some operation performed on those rows. To generate a single, final report would then require combining the results of those operations in a new or extended dataframe.

This sequence of operations is common enough for it to have been described as the 'split-apply-combine' pattern. The sequence is to:

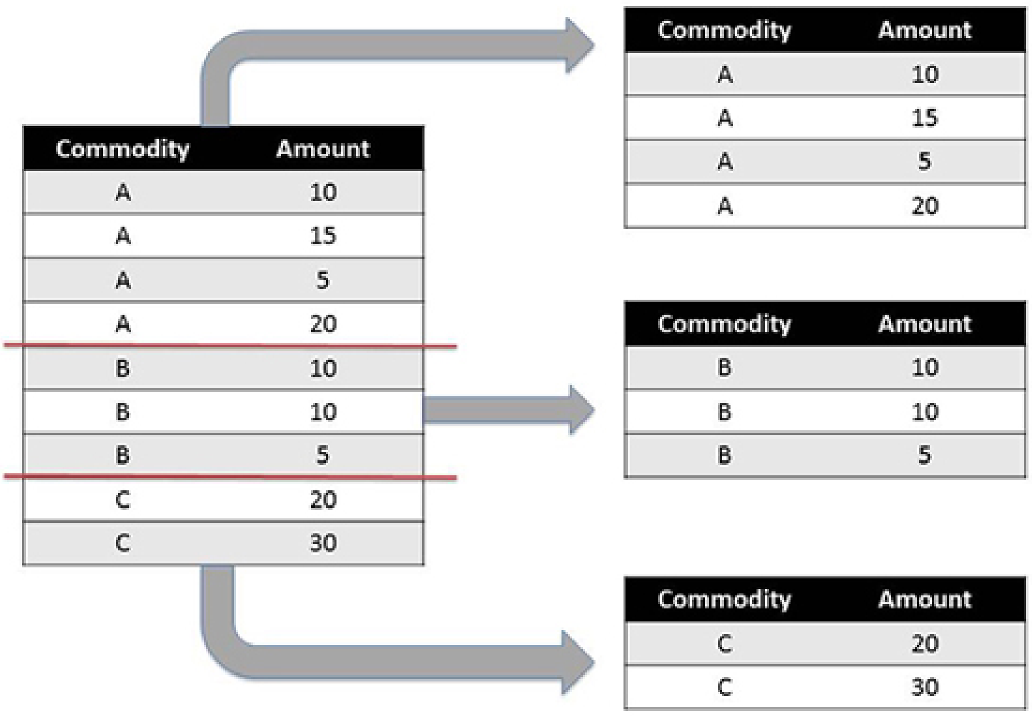
- 'split' an appropriately shaped dataset into several components
- 'apply' an operator to the rows contained within a component
- 'combine' the results of applying to operator to each component to return a single combined result.

You will see how to make use of this pattern using pandas next.

1.1 Splitting a dataset by grouping

'Grouping' refers to the process of splitting a dataset into sets of rows, or 'groups', on the basis of one or more criteria associated with each data row.

Grouping is often used to split a dataset into one or more distinct groups. Each row in the dataset being grouped around can be assigned to one, and only one, of the derived groups. The rows associated with a particular group may be accessed by reference to the group or the same processing or reporting operation may be applied to the rows contained in each group on a group by group basis.



Commodity	Amount
A	10
A	15
A	5
A	20
B	10
B	10
B	5
C	20
C	30

Commodity	Amount
A	10
A	15
A	5
A	20

Commodity	Amount
B	10
B	10
B	5

Commodity	Amount
C	20
C	30

Figure 2

The rows do not have to be ‘grouped’ together in the original dataset – they could appear in any order in the original dataset (for example, a commodity A row, followed by a two commodity B rows, then another commodity A row, and so on). However, the order in which each row appears in the original dataset will typically be reflected by the order in which the rows appear in each subgroup.

Let’s see how to do that in pandas. Create a simple dataframe that looks like the full table in the image above:

In []:

```
data=[['A',10],['A',15],['A',5],['A',20],
['B',10],['B',10],['B',5],
['C',20],['C',30]]
df=DataFrame(data=data, columns=["Commodity","Amount"])
df
```

Out[]:

	Commodity	Amount
0	A	10
1	A	15
2	A	5
3	A	20

4	B	10
5	B	10
6	B	5
7	C	20
8	C	30

Next, use the **`groupby()`** method to group the dataframe into separate groups of rows based on the values contained within one or more specified 'key' columns. For example, group the rows according to what sort of commodity each row corresponds to as specified by the value taken in the 'Commodity' column.

In []:

```
grouped = df.groupby('Commodity')
```

The number and 'names' of the groups that are identified correspond to the unique values that can be found within the column or columns (which will be referred to as the 'key columns') used to identify the groups.

You can see what groups are available with the following method call:

In []:

```
grouped.groups.keys()
```

Out []:

```
['A', 'C', 'B']
```

The **`get_group()`** method can be used to grab just the rows associated with a particular group.

In []:

```
grouped.get_group('B')
```

Out []:

	Commodity	Amount
4	B	10
5	B	10
6	B	5

Datasets can also be grouped against multiple columns. For example, if there was an extra 'Year' column in the above table, you could group against just the commodity, exactly as above, to provide access to rows by commodity; just the year, setting **`grouped = df.groupby( 'Year' );`** or by both commodity and year, passing in the two grouping key columns as a list:

```
grouped = df.groupby( ['Commodity','Year'])
```

The list of keys associated with the groups might then look like [('A', 2015), ('A', 2014), ('B', 2015), ('B', 2014)]. The rows associated with the group corresponding to commodity A in 2014 could then be retrieved using the command:

```
grouped.get_group( ('A',2014) )
```

This may seem to you like a roundabout way of filtering the dataframe as you did in Week 2; but you'll see that the ability to automatically group rows sets up the possibility of then processing those rows as separate 'mini-dataframes' and then combining the results back together.

Exercise 2 Grouping data

Work through Exercise 2 in the Week 4 notebook.

As you complete the tasks, think about these questions:

- For your particular dataset, how did you group the data and what questions did you ask of it? Which countries were the major partners of your reporter country for the different groupings?
- With the ability to group data so easily, what other sorts of questions would you like to be able to ask?

1.2 Looking at apply and combine operations

Having split a dataset by grouping, an operation is 'applied' to each group.

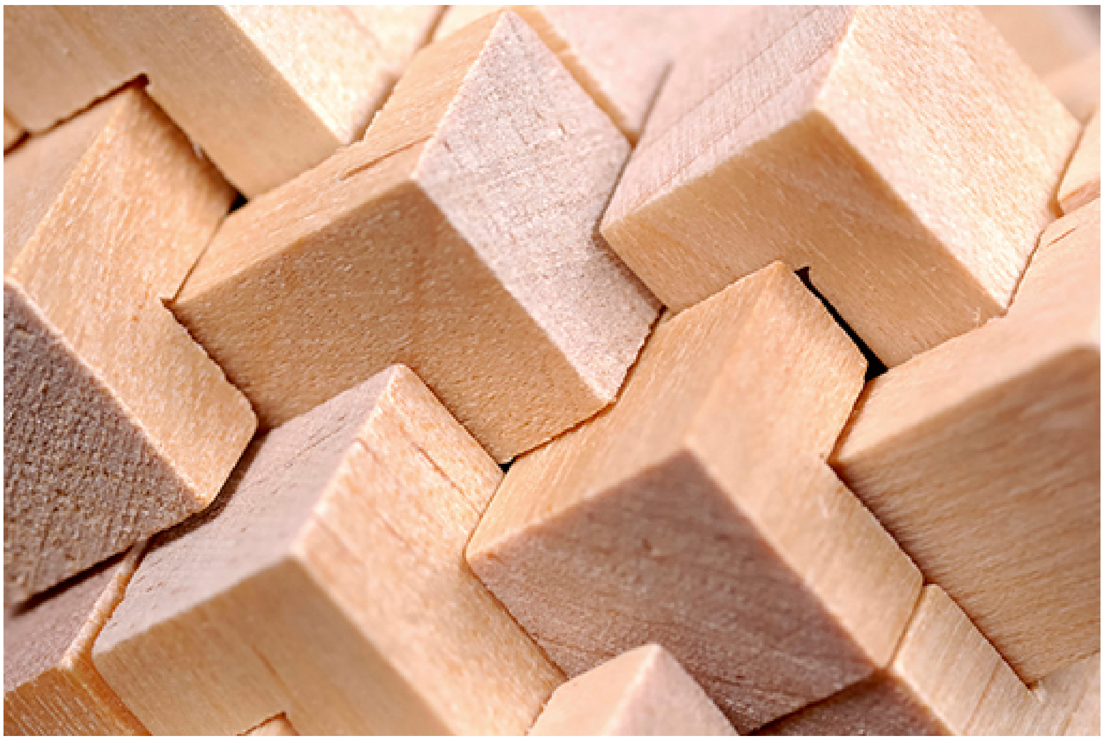


Figure 3

The operation often takes one of two forms:

- a 'summary' operation, in which a summary statistic based on the rows contained within each group is generated. A single value is returned for each group, for example, the group median or mean, the number of rows in the group, or the maximum or minimum value in the group. The final result will have *M* rows, one for each of the M groups created by the split (that is, . **`groupby()`**) operation.

- a ‘filtering’ or ‘filtration’ operation, in which groups of rows are retained or discarded based on a particular property of the group as a whole. For example, only groups of rows where the sum of all the values in the group is above some threshold are retained. The effect is that each group keeps the same number of rows, but the resulting dataset (after combination, see below) may contain fewer groups than the original.

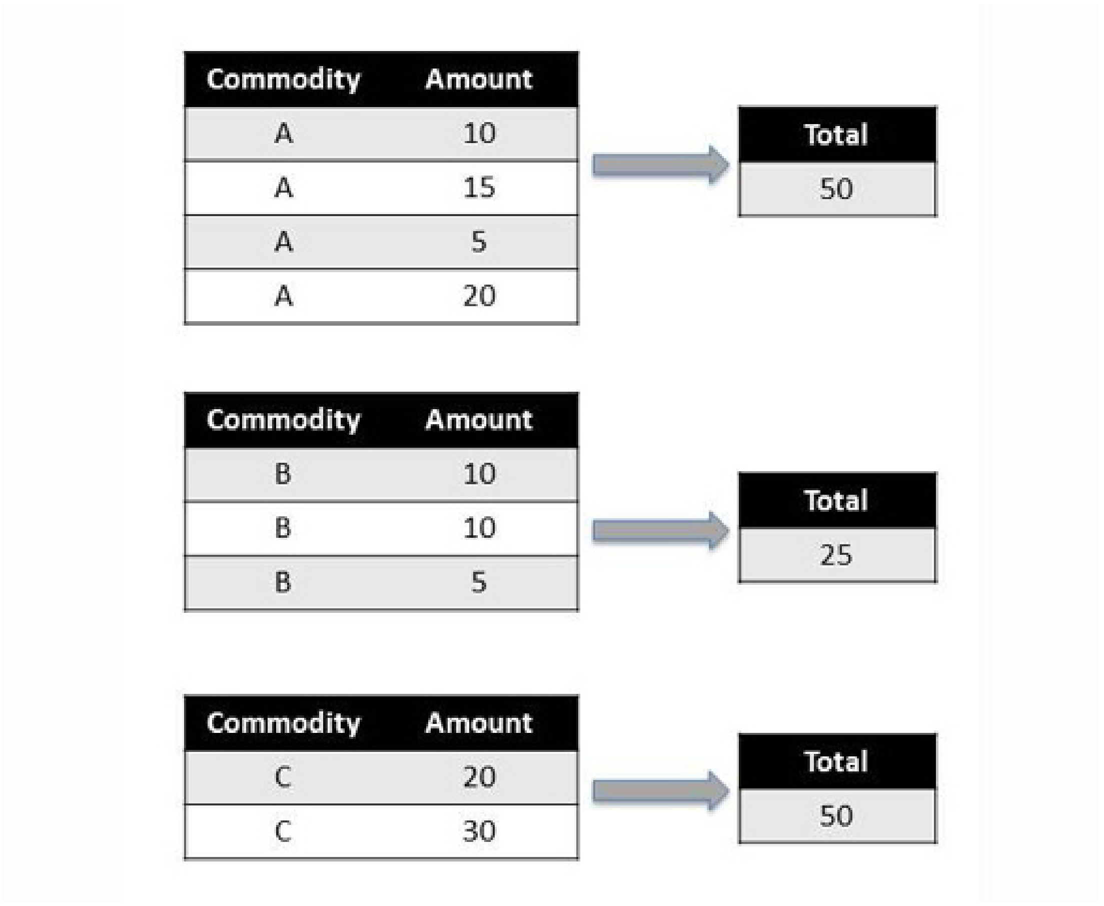
The results of applying the summary or filtration operation are then combined to provide a single output dataframe.

In the next section, you will see how to apply a variety of summary operations, and in a later step examples of filtration operations.

1.3 Summary operations

Summary, or aggregation, operations are used to produce a single summary value or statistic, such as the group average, for each separate group.

Find the ‘total’ amount within each group using a **summary** operation:



Commodity	Amount
A	10
A	15
A	5
A	20

Total
50

Commodity	Amount
B	10
B	10
B	5

Total
25

Commodity	Amount
C	20
C	30

Total
50

Figure 4

To apply a summary operator to each group, such as a function to find the mean value of each group, and then automatically combine the results into a single output dataframe, pass the name of the function in to the **`aggregate()`** method. Note that pandas will try to use this operator to summarise each column in the grouped rows separately if there is more than one column that can be summarised. So for example, if there was a ‘Volume’ column, it would also return total volumes.

Let’s use again the example dataframe defined earlier:

In []:

```
df
```

Out[]:

	Commodity	Amount
0	A	10
1	A	15
2	A	5
3	A	20
4	B	10
5	B	10
6	B	5
7	C	20
8	C	30

Group the data by commodity type and then apply the **`sum`** operation and combine the results in an output dataframe. The grouping elements are used to create index values in the output dataframe.

In []:

```
    grouped=df.groupby('Commodity')
    grouped.aggregate(sum)
```

Out[]:

	Amount
Commodity	
A	50
B	25
C	50

In this case, the **`aggregate()`** method applies the sum summary operation to each group and then automatically combines the results. For a **summary** operation such as this, the resulting combined dataframe contains as many rows as there were groups created by the splitting **`.groupby()`** operation.

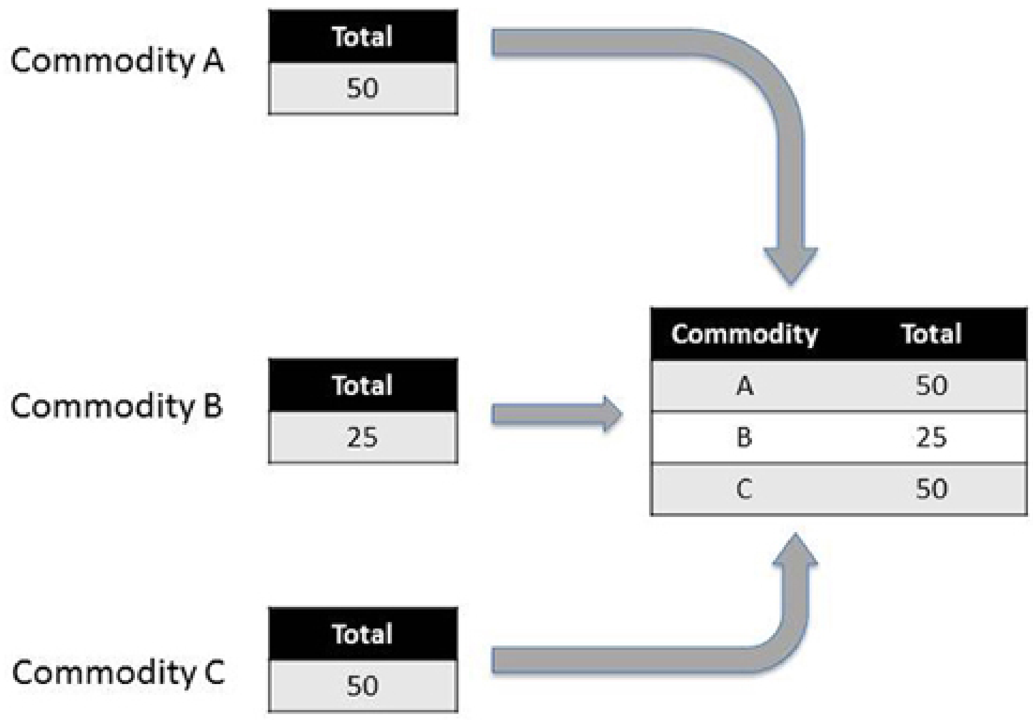


Figure 5

The slightly more general **`apply()`** method can also be substituted for the **`aggregate()`** method and will similarly take the rows associated with each group, apply a function to them, and return a combined result.

The **`apply()`** method can be really handy if you have defined a function of your own that you want to apply to just the rows associated with each group. Simply pass the name of the function to the **`apply()`** method and it will then call your function, once per group, on the sets of rows associated with each group.

For example, find the top two items by ‘Amount’ in each group:

`In []:`

```
def top2byAmount(g):
return g.sort_values('Amount', ascending=False).head(2)
grouped.apply(top2byAmount)
```

`Out[]:`

```
                Amount
Commodity
A            3  20
             1  15
B            4  10
             5  10
```

C	8	30
	7	20

The second index column containing the numbers 3, 1, 4 etc., contains the original index value of each row.

In Week 3 the **`apply()`** method was called on a column, to apply the given function to each cell. Here it was called on a grouped dataframe, to apply the given function to each group.

Exercise 3 Experimenting with split-apply-combine

Work through Exercise 3 in your Exercise notebook 4 to practise the summary operations.

As you complete the tasks, think about these questions:

- For your dataset, which months saw the highest and lowest levels of trade activity? Did there appear to be any seasonal behaviour?
- When graphically comparing total trade flows from the leading partner countries to the World total, did it look as if any partners particularly dominated that area of trade? If you have time, find news reports discussing why this should be the case.

1.4 Filtering groups

Being able to group rows according to some criterion and then apply various operations to those groups is a very powerful technique.

However, there may be occasions when you only want to work with a subset of the groups that can be extracted from a single dataset based on a particular group property. For example, it might require that:

- groups that contain a minimum number of rows, such as countries that engage in trade around a particular commodity with a minimum number of partner countries
- groups for whom a summary statistic meets certain conditions (for example, the total value of exports for a particular commodity exceeds a particular threshold value, or whose minimum or maximum value are below a certain value)
- a ranking of the groups based on a particular summary statistic, such as the total trade value, that returns only the top five or bottom three groups according to that ranking.

In the following example, where groups are selected based on group size, a filtering operation is applied to limit an original dataset so that it includes just those groups containing at least three rows, combining the rows from the selected groups back together again to produce the output dataset:

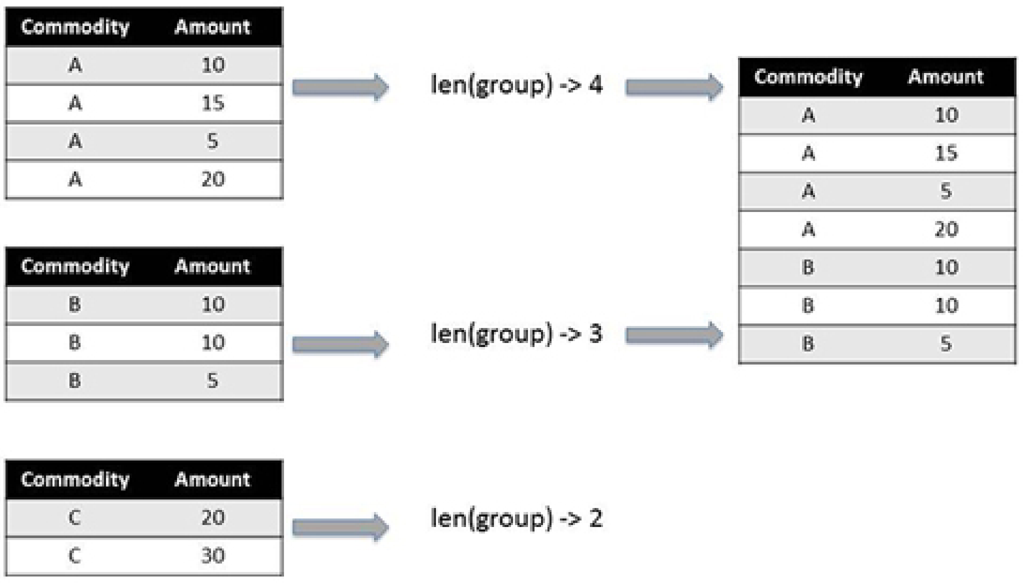


Figure 6

In pandas, groups can be filtered based on their group properties using the **`filter()`** method. Using the example dataframe again:

`In []:`

```
df
```

`Out[]:`

	Commodity	Amount
0	A	10
1	A	15
2	A	5
3	A	20
4	B	10
5	B	10
6	B	5
7	C	20
8	C	30

For example, the dataframe can be filtered to return just the rows from groups where there is a maximum number of rows in the group.

As a reference point, count how many rows are associated with each group.

`In []:`

```
grouped = df.groupby('Commodity')
```

```
grouped.aggregate(len)
```

Out[]:

	Amount
Commodity	
A	4
B	3
C	2

The **`filter()`** method uses a function that returns a boolean (**`True`** or **`False`**) value to decide whether or not to filter through the rows associated with a particular group.

As with the **`apply()`** method, provide the **`filter()`** method with just a function name in order to pass each group to that function. For example, define a function that says whether or not a group contains three or fewer rows and use that as a basis for filtering the original dataset.

In []:

```
def groupsOfAtMostThreeRows(g):
return len(g) <= 3
grouped.filter(groupsOfAtMostThreeRows)
```

Out[]:

	Commodity	Amount
4	B	10
5	B	10
6	B	5
7	C	20
8	C	30

Alternatively, all the rows in a group can be filtered on an aggregate property of the group such as the sum total, or maximum, minimum or mean value, from one of the columns.

In []:

```
#Consider the following total amounts by group
grouped.aggregate(sum)
```

Out[]:

	Amount
Commodity	
A	50

B	25
C	50

In []:

```
pivot_table(df,
index=['Commodity','Partner'],
values='Amount',
aggfunc=sum)
```

Out[]:

	Commodity	Amount
0	A	10
1	A	15
2	A	5
3	A	20
7	C	20
8	C	30

The ability to filter datasets based on group properties means that large datasets can more easily be limited to just those rows associated with groups of rows that are deemed to be relevant in some way.

Exercise 4 Filtering groups

Use the Exercise notebook 4 to practise filtering in Exercise 4.

As you complete the tasks, think about other questions you could ask of your data using the filter command.

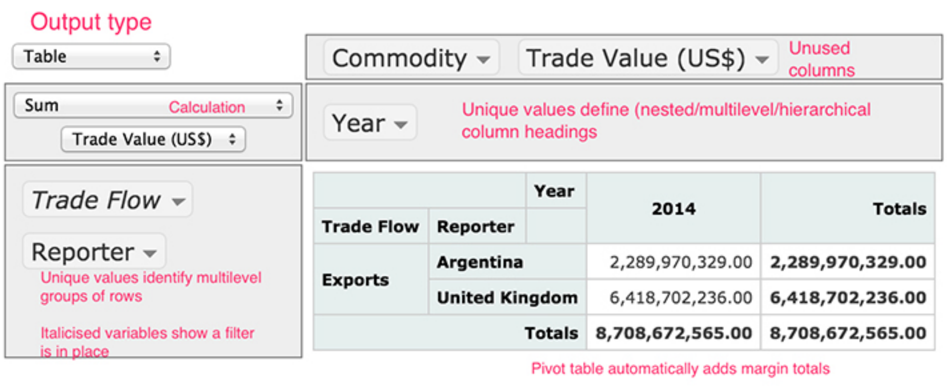
2 Pivot tables

One of the most useful, if poorly understood, features offered by many spreadsheet applications is the 'pivot table'.

Pivot tables provide a way of creating summary reports over particular parts of a dataset, reshaping the data into grouped rows, itemised columns, and summary values within each group and item.

The screenshot of the interactive pivot table shown below, based on a widget originally created by Nicolas Krutchen at Datacritic, contains a small fragment of the Comtrade data describing milk imports to the UK.

The pivot table is organised as follows:


Output type
Table
Commodity
Trade Value (US$)
Unused columns
Sum
Calculation
Trade Value (US$)
Year
Unique values define (nested/multilevel/hierarchical column headings
Trade Flow
Reporter
Unique values identify multilevel groups of rows
Italicised variables show a filter is in place
Pivot table automatically adds margin totals

	Year	2014	Totals
Trade Flow	Reporter		
Exports	Argentina	2,289,970,329.00	2,289,970,329.00
	United Kingdom	6,418,702,236.00	6,418,702,236.00
	Totals	8,708,672,565.00	8,708,672,565.00

Figure 7

You can see how the 'Trade Flow' and 'Reporter' columns are used to group the data, with each row representing a separate group. In addition, the values in the 'Year' column are broken out to create separate columns (although in this example there is only data for one year, and hence one 'Year' column, 2014). The function that is applied to the grouped data is a `sum` operation, and it is applied to the selected 'Trade Value (US$)' column in the original dataset. A marginal total value is calculated by summing across all the columns. The 'Commodity' and 'Trade Value (US$)' columns, while part of the original dataset, are not directly used to define the pivot table's structure; that is, they are not used to set the row or column index header labels in the displayed pivot table.

In terms of the split-apply-combine pattern, the pivot table operates as follows:

- the column names from the original dataframe that are listed in the rows panel on the left hand side of the interactive pivot table split the data into a set of groups, with each row specifying a group
- the pivot table's columns are set according to the unique values associated with the specified columns from the orignal dataframe; these break the data down into yet smaller groups that are associated with each cell.

The selected operator is then applied to each cell level group, the results combined and an appropriately structured output table is displayed.

To create a pivot table report for a dataset, typically three actions will be needed:

- identify what elements will appear as the row index values – that is, how the rows will be grouped. Typically, groups will be created based on the unique values within a single column or a combination of values, one from each of multiple grouping columns.

- identify what elements will appear as column headings. Again, the column heading may just be the unique values of a single variable, or combined values across multiple grouping columns.
- identify what numbers will be reported on. This step may often break down into two smaller steps:
  - to count the number of rows associated with a particular combination of row and column index values, select the count operation
  - to perform an operation on the value of cells in another column, select that column and then identify what operation to apply to it. For example, find the sum or mean values of a numerical quantity associated with rows keyed by the row and column index values, or count the number of unique values of a particular variable in rows identified by those key values.

In addition, one or more 'filters' can be added to the selection of row and column index values, either limiting which unique values in each key column to report on, or, by default, selecting them all.

It is often easier to understand how a pivot table is organised by using it interactively. You'll get a chance to do this in the next exercise.

Exercise 5 Interactive pivot table

If you haven't already, open the comtrade_pivot.html and save it into the same folder as the Exercise notebook 4. Then either re-run all the notebook cells, or just run the cell that contains the interactive pivot table.

Configuring a pivot table requires paying careful attention to the selection of row (grouping) values, columns (reported values) and summary (aggregating) function.

How easy did you find it to use the interactive pivot table? Could you work out how to select the row and column labels in order to ask particular questions of the data? What sorts of questions did you try to ask?

2.1 Pivot tables in pandas

The interactive pivot table provides a convenient way of exploring a relatively small dataset directly within a web browser. (A python package is also available that allows interactive pivot tables to be created directly from a pandas dataframe.)

Figure 8

You can also achieve a similar effect using code, one-line-at-a-time. In this step, you will learn how to ask – and answer – questions of a similar form to the ones you raised using the interactive pivot table, but this time using programming code.

There are several reasons why you might want to automate pivot table operations you might previously have done by hand. These include:

- having a record of all the steps used to perform a particular task, or analysis, which can be useful if you need to check or provide evidence about what you have done (transparency)
- being able to repeat the task automatically; this is particularly useful if you need to perform the same task repeatedly – for example, generating a new summary report each time a dataset is updated with new weekly or daily figures
- being able to apply one analysis to another dataset. For example, you might want to produce the same sort of pivot table reports to similarly organised datasets but differently populated datasets (for example, Comtrade datasets that refer to different groups of countries and/or different commodity types).

In order to use the interactive pivot table, you had to identify:

- what column(s) in the dataset to use to define the row groupings in the pivot table
- what column(s) in the dataset to use to define the column groupings in the pivot table
- what column in the dataset to use as the basis for the pivot table summary function
- what summary function to use.

The process is similar when it comes to using pivot tables in pandas. Indeed, you might find it useful to use the interactive pivot table to help you identify just what needs to go where in order to generate a particular report using the pandas pivot table.

Working with pandas pivot tables

Let’s start by creating a sample dataset that includes several different columns that can be grouped around. The code below defines the dataframe column by column, instead of row by row as you have learned before.

In []:

```
df = DataFrame({"Commodity":["A","A","A","A","B","B","B","C","C"],
"Amount":[10,15,5,20,10,10,5,20,30],
"Partner":["P","P","Q","Q","P","P","Q","P","Q"],
"Flow":["X","Y","X","Y","X","Y","X","X","Y"]})
df
```

Out[]:

	Commodity	Partner	Flow	Amount
0	A	P	X	10
1	A	P	Y	15
2	A	Q	X	5
3	A	Q	Y	20
4	B	P	X	10
5	B	P	Y	10
6	B	Q	X	5
7	C	P	X	20
8	C	Q	Y	30

Suppose, for example, that you have data for a particular reporter country, and that you want to find the total value of trade that country has for each commodity and each partner country. A pivot table can be used to split the data by ‘commodity’, and within that ‘partner’, and then apply some sort of aggregation function to each (‘commodity’, ‘partner’) group.

In the interactive pivot table, this would have meant ordering the ‘Commodity’ and ‘Partner’ labels in the rows area, setting the aggregation function to `sum` and applying it to the ‘Amount’ (that is, the ‘Trade Value’), and leaving the columns area free of any selections.

In turn, the pandas `pivot_table()` function uses:

- the `index` parameter set as a list containing the ‘Commodity’ and ‘Reporter’ data elements, to define the row categories
- the `values` parameter set to ‘Amount’
- the `aggfunc` (aggregating function) set to `sum` .

In []:

```
pivot_table(df,
index=['Commodity','Partner'],
values='Amount',
aggfunc=sum)
```

Out[]:

	Flow	X	Y
Commodity	**Partner**		
A	P	10	15
	Q	5	20
B	P	10	10
	Q	5	NaN
C	P	20	NaN
	Q	NaN	30

To further subdivide the data, an additional ‘Flow’ grouping element could be added in. (In this case, the resulting pivot table just corresponds to the original dataset.)

In []:

```
    pivot_table(df,
    index=['Commodity','Partner','Flow'],
    values='Amount',
    aggfunc=sum)
```

Out[]:

Commodity	Partner	Flow	
A	P	X	10
		Y	15
	Q	X	5
		Y	20
B	P	X	10
		Y	10
	Q	X	5
C	P	X	20
	Q	Y	30

Alternatively, you might decide that you want to pull out the ‘Flow’ items into separate columns for each of the original (‘commodity’, ‘partner’) groupings. To do this, add in a columns parameter:

```
pivot_table(df,
index=['Commodity','Partner'],
columns=['Flow'],
values='Amount',
aggfunc=sum)
```

	Flow	X	Y
Commodity	**Partner**		
A	**P**	10	15
	Q	5	20
B	**P**	10	10
	Q	5	NaN
C	**P**	20	NaN
	Q	NaN	30

In this case, some missing values arise for cases where there was no original row item. For example, there were no rows in the original dataset for Commodity/Partner/Flow values of B/Q/Y, C/P/Y or C/Q/X.

The pandas produced pivot table can be further extended to report 'marginal' items, that is, row and column based total amounts, by setting **`margins=True.`**

```
pivot_table(df,
index=['Commodity','Partner'],
columns=['Flow'],
values='Amount',
aggfunc=sum,
margins=True)
```

	Flow	X	Y	All
Commodity	**Partner**			
A	**P**	10	15	25
	Q	5	20	25
B	**P**	10	10	20
	Q	5	NaN	5
C	**P**	20	NaN	20
	Q	NaN	30	30
All		50	75	125

In terms of the ‘split-apply-combine’ pattern, the pandas pivot table operates in much the same way as the interactive pivot table:

- the list of original data columns assigned to the index parameter splits the data into a set of groups
- the groups are further split into smaller cell level groupings by optionally setting the columns parameter.

The selected operator is then applied to each group and the results combined in an appropriately structured output display table.

Exercise 6 pivot tables with pandas

Use the Exercise notebook 4 to explore the creation of pivot tables using pandas in Exercise 6.

Did you manage to ask any new questions of your data using the pandas pivot table function? You could try using them in combination with other pandas functions, such as **`filter()`** , to limit the rows you generated the pivot table against. What did the pivot tables tell you about the levels of trade around the trade item and reporter country you selected?

One reason that pivot tables are often thought of as difficult to use is that there is a lot of data manipulation going on inside them. The data is grouped across rows, split across columns and may be aggregated in various ways. It can sometimes be hard to work out how to structure the output report you want, even before worrying about the programming code syntax. Given that, consider what you think the benefits of using code are as opposed to interactive pivot tables. Think about how you could use them to complement each other.

2.2 Looking at the milk and cream trade

This week’s project looks at the milk and cream trade between the UK and other countries in the first five months of 2015.

Figure 9

The written up analysis is in the project notebook. You will also need to open the file comtrade_milk_uk_jan_jul_15.csv and save it to your Anaconda folder or CoCalc project.

The structure is very simple: besides the introduction and the conclusions, there is one section for each research question.

Extend or create your own project next.

2.3 Your project

If you have time, extend my project to answer different questions or create your own project.

Figure 10

Activity 1 Extend the project

Make a copy of the project notebook and change it to answer one or all of the following questions:

- Which are the regular exporters, i.e. which countries sell every month both unprocessed and processed milk and cream to the UK?
- Where could the export market be further developed, i.e. which countries import the least? Do the figures look realistic?
- What is total amount of exports to and imports from the bi-lateral trade countries? Hint: pivot tables can have ‘marginal’ values.
- Repeat the whole analysis for January–May 2014 and compare the results.

Activity 2 Create a project (optional)

If you have more time, create a completely new project. You could choose completely different commodities, a different reporter (e.g. your country), a different period (e.g. two or more full years), and only a few select partners (e.g. just the ‘World’ partner for a global analysis).

3 This week's quiz

Now it's time to complete the Week 4 badge quiz. It is similar to previous quizzes, but this time instead of answering five questions there will be fifteen.

Week 8 compulsory badge quiz

Remember, this quiz counts towards your badge. If you're not successful the first time, you can attempt the quiz again in 24 hours.

4 Summary

Phew – you made it! Well done!

Figure 11

During this week you've learned how to take a dataset that contains multiple possible groupings or subsets of data, and work with those groups to perform a variety of transformations.

In particular, you have learned how to:

- split the data contained in a dataframe into multiple groups based on the unique 'key' values in a single column, or unique combinations of values that appear across two or more columns
- **`apply`** an **`aggregate`** (summary) function to generate a single summary result for the group, and then combine these results to generate a summary report with as many rows as there were groups, one summary report row per group **`-apply`** a **`filter`** function that would use the rows contained in each group as the basis for a filtering operation, returning rows from each group who's group properties matched the filter conditions
- use a pivot table to generate a variety of summary reports.

You may not have thought of performing gymnastics with data before, but as you've seen, we can start to work our data quite hard if we need to get it into shape!

4.1 Week 7 and 8 glossary

Here are alphabetical lists, for quick look up, of what this week introduced.

Concepts

An **API** , or **application programming interface** provides a way for computer programmes to make function or resource requests from a software application. Many online applications provide a **web API** that allows requests to be made over the internet using web addresses or **URLs** (uniform resource locator). A URL may include several parameters that act as arguments used to pass information into a function provided by the API. To prevent ambiguity, simple punctuation is avoided in URLs. Instead, 'websafe' encodings using the ASCII encoding scheme are typically used to describe punctuation characters.

The notion of **grouping** refers to the collecting together of sets of rows based on some defining characteristic. Grouping on one or more key columns splits the dataset into separate groups, one group for each unique combination of values that appears in the dataset across the key columns. Note that not all possible combinations of cross-column key values will necessarily exist in a dataset.

The **split-apply-combine** pattern describes a process in which a dataset is **split** into separate groups, some function is **applied** to the members of each separate group, and the results then **combined** to form an output dataset.

Functions and methods

`df.to_csv(filename,index=False)` writes the contents of the dataframe `df` to a CSV file with the specified filename in the current folder. The `index` parameter controls whether the dataframe index is included in the output file.

`read_csv(URL,dtype={})` can be used to read a CSV file in from a web location given the web address or URL of the file. We also made use of an additional parameter, `dtype` to specify the data type of specified columns in a dataframe created from a CSV file.

`df.groupby(columnName)` or `df.groupby(listOfColumnNames)` is used to split a dataframe into separate groups indexed by the unique values of `columnName` or unique combinations of the column values specified in the `listOfColumnNames.`

`grouped.get_group(groupName)` is used to retrieve a particular group of rows by group name from a set of grouped items.

`grouped.groups.keys()` is used to retrieve the names of groups that exist within a set of grouped items.

`grouped.aggregate(operation)` applies a specified operation to a group (such as sum) and then combines the results into a single dataframe indexed by group.

`grouped.apply(myFunction)` will apply a user defined function to the rows associated with each group in a set of grouped items and return a dataframe containing the combined rows returned from the user defined function.

`grouped.filter(myFilterFunction)` will apply a user defined filtration function to each group in a set of grouped items that tests each group and returns a Boolean True or False value to say whether each group has passed the filter test. The `.filter()` function then returns a single dataframe that contains the combined rows from groups that the user defined filter function let through.

`pivot_table(df, index=indexColumnNames, columns=columnsColumnNames, values=valueColumnName, aggfunc=aggregationFunction)` generates a pivot table from a dataframe using unique combinations of values from one or more columns specified by the `indexColumnNames` list to define the row index and unique combinations of values from one or more columns specified by the `columnsColumnNames` list to define the columns. The pivot table cells are calculated by applying the `aggfunc` function to the `valueColumnName` column in the group of rows associated with each cell.

4.2 What next?

Video content is not available in this format.

As the course comes to an end, what's next in your learning journey? The National Careers Service can help you decide your next steps with your new skills. Ruth also mentions extending your learning by investigating more open data.

Exploring open data further

The last few years has seen a wide variety of local and national governments and agencies publishing data as 'open data' that can be freely re-used by anyone. Explore some of this data yourself, at the following links:

- UK government open data site – a directory of UK public datasets
- US government open data site – the home of the US Government's open data
- Open Knowledge Global Open Data Index – a comprehensive directory of national open data initiatives
- Open Data Inception – a geographic list of over 1500 data portals around the world
- Google Public Data Explorer – a further list of data providers, with charts for some datasets
- Many towns and cities also have their own data sites: search for the name of your town and the keywords 'open data store'
- Open data published by government departments and agencies such as performance of UK schools or prices paid for house sales in the UK
- The pandas library supports a growing number of external data sources such as Google Analytics.

Get careers guidance

The National Careers Service can help you decide your next steps with your new skills.

Complete our survey

We would love to know what you thought of the course and what you plan to do next. Whether you studied the course all in one go or dipped in and out, please take our end-of-course survey. Your feedback is anonymous but will help us to improve what we deliver.

Tell us what you think

Now you've come to the end of the course, we would appreciate a few minutes of your time to complete this short end-of-course survey (you may have already completed this survey at the end of Week 4).

Additionally, if you found this course through the Skills Toolkit launched by the UK government in April 2020 and would be willing to provide feedback on how this course has helped you, please get in touch by emailing us.

Acknowledgements

This free course was written by Michel Wermelinger.

Images

Course image © peterhowell/iStockphoto.com
Figure 1 © AFP/Getty Images
Figure 2 © Savushkin/istockphoto.com
Figure 3 © Kameleon007/istockphoto.com
Figure 4 © Picardo/Getty Images.co.uk
Figure 5 © kerriekerr/istockphoto.com
Figure 6 © Rawpixel Ltd/istockphoto.com
Figure 7 © merc67/istockphoto.com
Figure 8 © Katherine Feng/Globio/Getty Images
Figure 9 © AFP/Getty Images
Figure 10 © Nicolas Raymond in Flickr https://creativecommons.org/licenses/by/2.0
Figure 11 created by Jonatan argento in Wikipedia, https://creativecommons.org/licenses/by-sa/2.5/deed.en
Figure 12 © Fuse/Getty Images
Figure 13 © Predrag Vuckovic/Getty Images
Figure 1 © David Sucsy/istockphoto.com
Figure 2 © santiphotois/istockphoto.com
Figure 3 © Fuse/Getty Images
Figure 4 © Biddiboo/Getty Images
Figure 5 © maximkabb/istockphoto.com
Figure 6 © D-BASE/Getty Images
Figures 1 and 2 © Igor Zhuravlov/via istockphoto.com
Figure 3 © MartinCParker/istockphoto.com
Figure 4 © Adri Berger/Getty Images
Figure 5 © Donald Iain Smith/Getty Images

Figure 6 Public Domain by US Federal Government https://commons.wikimedia.org/wiki/File:Compass_rose.png

Figure 7 © Jim Reed/Getty Images

Figure 1 © David Sucsy/istockphoto.com

Figure 2 © santiphotois/istockphoto.com

Figure 3 © Fuse/Getty Images

Text

Exercise notebook 3 and Exercise 1: and other identifiable exercises and tables compiled using The World Bank data (2013).

Figure 1 © Biddiboo/Getty Images

Figure 2 © maximkabb/istockphoto.com

Figure 3 © D-BASE/Getty Images

Figure 6 © Uwe Kreijci/Getty Images

Figure 7 © Ariel Skelley/Getty Images

Figure 1 © cgouin/istockphoto.com

Figure 2 © Sam Camp/istockphoto.com

Figure 3 © Terribil-T/istockphoto.com

Figures 4 and 5 adapted from Comtrade data extraction interface: https://comtrade.un.org/data/

Figure 6 © ronstik.istockphoto.com

Figure 8 © webking/istockphoto.com

Figure 13 pivot table originally created by Nicolas Krutchen at Datacritic containing some Comtrade data describing milk imports to the UK

Figure 14 © Hung_Chung_Chih/istockphoto.com

Figure 15 © Jack Andersen/Getty Images

Figure 16 © PeopleImages/istockphoto.com

Figure 17 © oleg66/istockphoto.com

Text

Activity 3 Project: compiled from World Health Organisation (WHO) information)

Video

Exercise 1 video: © The Open University

Every effort has been made to contact copyright owners. If any have been inadvertently overlooked, the publishers will be pleased to make the necessary arrangements at the first opportunity.