



german
cooperation

DEUTSCHE ZUSAMMENARBEIT

Implemented by

giz

Deutsche Gesellschaft
für Internationale
Zusammenarbeit (GIZ) GmbH



The Data Science in Python Playbook

How to turn data into assets



Data Science in Python Playbook

How to turn data into assets

Contents

1.0	How to Get Started	6
2.0	Introduction - What is Python Built on?	8
	Numbers	12
	Strings	13
	A Brief Introduction to Methods	15
	Lists, Tuples, and Dictionaries	17
	Lists	17
	Accessing Values in Lists	18
	Updating Lists	18
	Tuples	20
	Dictionaries	20
	Loops	23
3.0	Libraries (Otherwise Known as Modules).....	27
4.0	Functions	33
5.0	The DataFrame.....	36
	Pandas Series.....	37
	Head and Tail	38
	Dealing with Missing Data	38
	Data Cleaning - Worked Example.....	41
6.0	Indexing and Selecting Data in Pandas	44
	[]	45
	.loc	45
	.ilo	46
	Worked Examples	46
7.0	Aggregation	52

8.0 Plotting.....56

 Basic - Series Plots57

 Dataframe Plots.....60

9.0 Introduction to Machine Learning Models64

**10.0 Can We Make a Model That Accurately Predicts the Amount
of Tea Plucked in Advance?.....70**

 Weather Data71

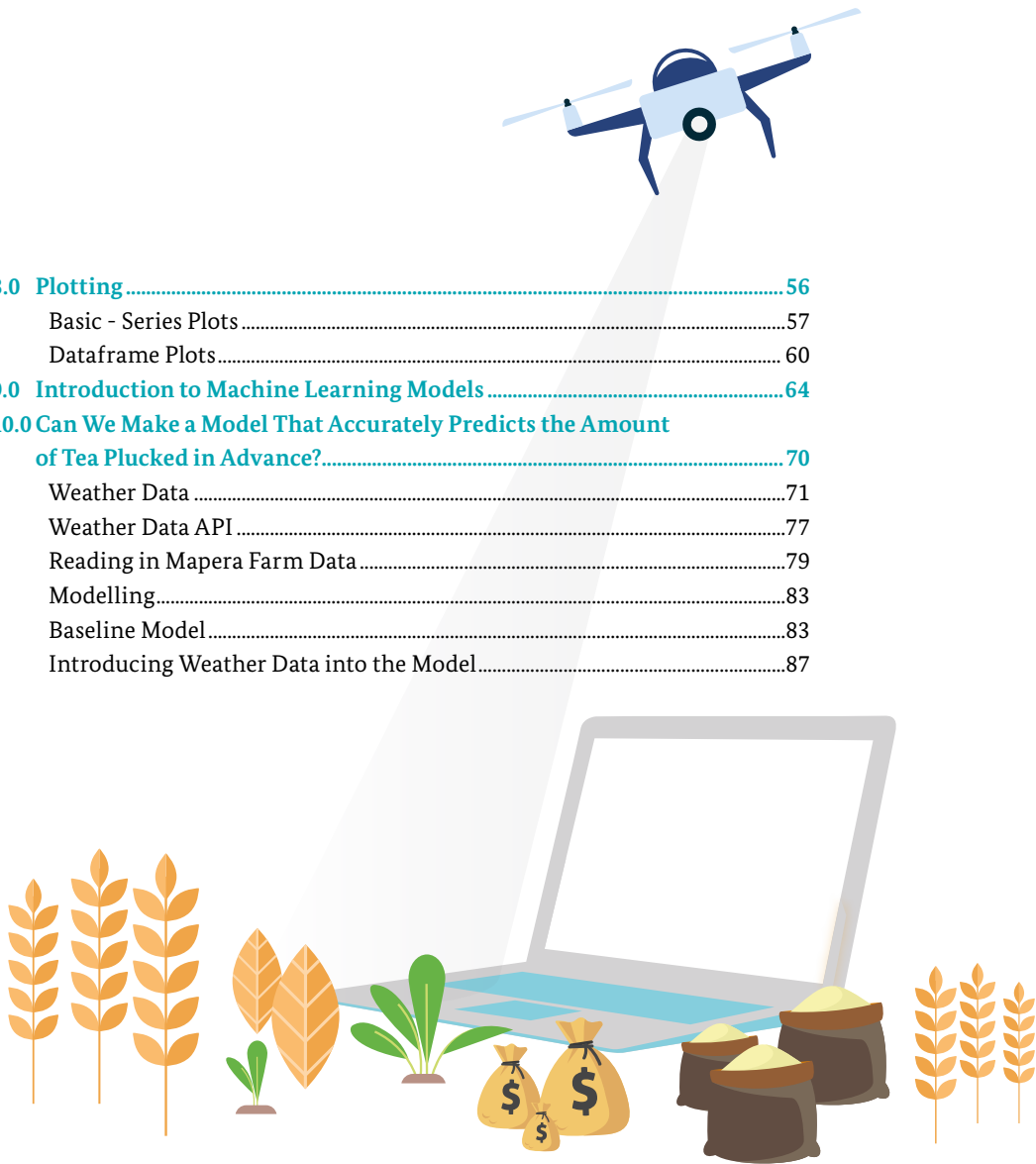
 Weather Data API77

 Reading in Mapera Farm Data79

 Modelling.....83

 Baseline Model.....83

 Introducing Weather Data into the Model.....87



1.0

How to Get Started



How to Get Started

In order to follow this playbook and be able to conduct the containing Python exercises directly in Jupyter Notebook, please make sure to install the following components on your local machine before getting started:

- Install Anaconda:

<https://www.anaconda.com/products/individual>

- YouTube video with installation instructions:

<https://www.youtube.com/watch?v=T8wK5loXkXg>

- Get a GitHub account:

<https://github.com/>



2.0

Introduction



INTRODUCTION - WHAT IS PYTHON BUILT ON?

Objects:

- An object is just another way of saying ‘this is a specific type of data and I can do certain things with it’.
- The purpose of writing code is to allow these objects to interact with each other in some way.
- Informally speaking, Python lets us ‘do things with stuff’, where the ‘stuff’ are objects, and the ‘things’ are the interactions, like: addition, multiplication, concatenation and anything else.
- From this we can say that to be fluent in Python is to have a strong understanding of what type of object you are working with, and how these objects can interact with each other.
- Objects are so important that when you download Python, it comes with built-in types of objects (usually just referred to as ‘types’) which are so common that the developers decided they should always be available to everyone.

NUMBERS (Int, Float, Decimal, Fraction)	STRINGS	LISTS	DICTIONARIES	TUPLES
5, 0.0, 0i+0j	'Hi', "Hello"	['one', 1]	{'Greet': 'Hallo'}	('one', 2)

Note that this not an exhaustive list of types, and their formal definition allows for much more than this, but practically speaking, these are the big ones that will come in handy when trying to code on a day-to-day basis.



Writing Comments in Python

In addition to writing code, note that it's always a good idea to add comments to your code. It will help others to understand what you were trying to accomplish (the reason why you wrote a given snippet of code). Not only does this help other people to understand your code, it can also serve as a reminder to you when you come back to it weeks or months later.

To write comments in Python, use the number symbol # before writing your comment. When you run your code, Python will ignore everything past the # on a given line.

Practice writing comments

```
>>> print("Hello, Python!") # This line prints a string
>>> # print("Hi")
```

After executing the cell above, you should notice that the second line did not appear in the output, because it was a comment (and thus ignored by Python). The second line was also not executed because print("Hi") was preceded by the number sign (#) as well! Since this isn't an explanatory comment from the programmer, but an actual line of code, we might say that the programmer 'commented out' that second line of code.

Errors in Python

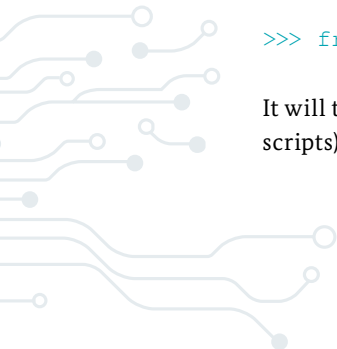
Everyone makes mistakes. For many types of mistakes, Python will tell you that you have made a mistake by giving you an error message. It is important to read error messages carefully to really understand where you made a mistake and how you may go about correcting it.

For example, if you spell print as frint, Python will display an error message. Give it a try:

Print string as error message

```
>>> frint("Hello, Python!")
```

It will tell you where the error occurred (more useful in large notebook cells or scripts), and what kind of error it was (NameError).



Here, Python attempted to run the function 'frint', but could not determine what 'frint' is since it's not a built-in function and has not been previously defined by us either.

You'll notice that if we make a different type of mistake, by forgetting to close the string, we'll obtain a different error (i.e. a `SyntaxError`). Try it below:

Try to see built-in error message

```
>>> print("Hello, Python!")
```

Does Python know about your error before it runs your code?

Python is what is called an interpreted language. Compiled languages examine your entire programme at compile time and can warn you about a whole class of errors prior to execution. In contrast, Python interprets your script line-by-line as it executes it. Python will stop executing the entire programme when it encounters an error (unless the error is expected and handled by the programmer, a more advanced subject that we'll cover later on in this course).

Try to run the code in the cell below and see what happens:

Print string and error to see the running order

```
>>> print("This will be printed")
>>> frint("This will cause an error")
>>> print("This will NOT be printed")
```

Built-in types can be combined in intuitive ways to let us build what we need.



Numbers

Let's take the example that we want to calculate how much tax we need to pay for the tea that has been picked on our farm. If we know:

- number of kilos plucked
- tax rate per kilo of tea

We can calculate the total amount of tax using the equation `tax = number of kilos plucked * tax rate per kilo of tea`.

If we know we picked 45 kilos of tea, and are taxed at Kshs 10 per kilo, we can put this into a cell and it will calculate the answer:

```
In [15]: 45 * 10
Out[15]: 450
```

When we typed 45 and 10, Python immediately knew that this was a `number` type (specifically it was an `int` (integer) type) which meant that it knew that you could use the `*` symbol to multiply these together.

Number types support the normal operations:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `**` exponentiation

We can also check to see what type our number is by using the function, which we'll explain soon:

`type()`

```
In [1]: type(45)
Out[1]: int
```



Great! So, with this knowledge we already know that we can use Python as a scientific calculator, doing arbitrarily complex calculations:

```
In [7]: M (5000 * (1.02)**(10) / (100 + 30) ) - 5
Out[7]: 41.88440076902913
```

Notice that brackets by themselves are also used in the way you would expect them to work in normal maths.

Strings

Strings are Python's name for text. This is how text is stored. For Python to recognise that you are trying to write a string, just write some text and surround that text with either single ' or double " quotation marks.

If you want Python to recognise that you are writing a string that spans multiple lines, use three quotation marks, either ''' or """ on each side of your text.

```
In [12]: M 'Hello, this works!'
Out[12]: 'Hello, this works!'

In [13]: M "This works too!"
Out[13]: 'This works too!'

In [16]: M '''So Does This'''
Out[16]: 'So Does This'
```

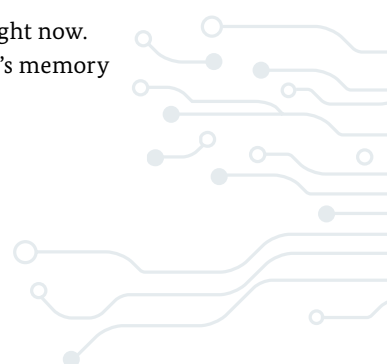
Again, we can check the type of the string to see that everything is working as expected.

```
In [17]: M type('Hello, this works!')
Out[17]: str
```

Remember, that these strings are different to what you are reading right now. These strings are objects which means that they are stored in Python's memory somewhere.

Strings also have the benefit of understanding operations:

- + addition of stings
- - multiplication, of a `str` and `int` type



Other operators that are understood as `number` types are not understood here, since they don't really make sense in the same way as the:

- `-` subtract operator
- `/` divide operator
- `**` exponentiation operator

```
In [20]: H 'Hello ' + 'would this work?'
Out[20]: 'Hello would this work?'

In [27]: H 'Hello' - 'what'

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-27-3db7e9cab44e> in <module>
----> 1 'Hello' - 'what'

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

An error is thrown here since Python does not understand how to subtract two strings from each other.

```
In [24]: H 'Hello' * 10
Out[24]: 'Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello '
```

Here, we've managed to create something that looks a lot like the copy-paste function.

Since we know that `strings` are a certain `type` of `object`, they all share certain functionalities which we may find handy in different situations.

Try pressing the 'full stop' key and the 'tab' key after you've written a string and see what comes up. You should see a list of things you can do out-of-the-box to a string.



Each item that has come up in the list is known as a `'method'` – we'll explain this later. For now, after you pick a method, make sure to type `()` after the word you selected for it to do something!

```
In [45]: M 'hello'.capitalize()
```

```
Out[45]: 'Hello'
```

```
In [48]: M 'hello'.upper()
```

```
Out[48]: 'HELLO'
```

```
In [50]: M 'hello'.find('h')
```

```
Out[50]: 0
```

A Brief Introduction to Methods

Before going any further, it is worth introducing the concept of a `method`.

We just used a method when working with strings above, when we wrote the code: `'hello'.capitalize()`. The general thing that we did here was call a method on an object. When written in code, it always looks the same:

`object.method(arguments)`

Where:

- **Object**: Is what we have been discussing so far and can be a numeric, string, and list type of object.
- **.**: The full stop is what we use to tell Python that the next thing we write will be a method, so it is important!
- **Method**: A piece of functionality that will save us from having to code the solution up ourselves.
- **Arguments****: Not introduced in the `'hello'.capitalize()` example. Since the method takes no arguments, this is a value that is required for the method to be able to work, i.e. a variable that must be set before the method can run. An example of this is the `'hello'.find('h')` example above. The method `'find'` requires a value to find, otherwise... what is it supposed to find?



This is a very common pattern which provides a lot of functionality to us without having to do very much work. This is a deep topic which goes to the very heart of the Python programming language itself, but we can think of it simply as **providing us with functionality that would take a long time to code ourselves**. (Capitalising a string may be quick, but there are some methods which save a huge amount of time, especially when we use lots of them together).

The other thing we have to know is that every object has its own set of methods. This is useful information because we know we can call the `capitalize()` method on any string.

There are some special and simple methods that were deemed so fundamental that they can be called without an object. There are not many of these, though some examples include:

- `print()`: Probably the most common method of its kind, is used all the time to print text, to debug, to explain, etc.
- `type()`: Checks what type of object the value in the argument is.
- `len()`: Checks the length of the object inside the brackets.
- `min()`: Returns the minimum value of a sequence of numbers.
- `max()`: Returns the maximum value of a sequences of numbers.
- `list()`, `dict()`, `tuple()`, `str()`, `float()`, `int()`: All the core types have a base method which converts a value to this object.
- `input()`: Type an input value from the output console for the code to run.

Knowing this information, now try using the `print()` built-in method to print your name, which has also been capitalised using the `.capitalize()` object method.

```
In [1]: # enter code here
```



Lists, Tuples, and Dictionaries

These three `types` are all examples of `containers`. All this means is that they are used to store other bits of data in various ways.

These containers can store any `object` and can be any length (as long as you have enough RAM), meaning that they are extremely flexible within their own definitions.

Lists

For lists, objects are stored in between square brackets `[]` and individual objects are separated by a comma `,`. These objects can be anything – even other lists! Operations available to lists:

- `+` Addition: Is only available for two lists where the symbol joins them together, like in a string.
- `*` Multiplication: Is like the string example only it works with an integer.
- `[i]` Slicing: Is the ability to select a single element or a range of a list and work with that.

Indexing

For containers (or any object that contains more than one object) we need to know how to select each element. Python's convention is to call the first element in any list, tuple, or dictionary the `0th` element. I.e. to select the first element in a list, we would use the number 0 inside square brackets:

```
List = ['first_item', 'second_item', 'another_item', 'final_item']
```

```
Index = 0 1 2 3
```

Example: `List[0] = 'first_item'`

```
In [59]: M ['hello', 3, "Yes", ['Another', 'list?'], None, 10] + [3]
```

```
Out[59]: ['hello', 3, 'Yes', ['Another', 'list?'], None, 10, 3]
```

```
In [62]: M [5,10] * 3
```

```
Out[62]: [5, 10, 5, 10, 5, 10]
```



Accessing Values in Lists

In order to only access 'Pruning' from the list below, select the list object along with the index of the value that you would like to select.

An important thing to note is that the index of a list begins with 0 instead of 1. This means that if you wanted to select the value 'Tea' from the activities list below, the correct syntax is: `activities[0]`

```
In [36]: M activities = ['Tea', 'Plucking', 'Pruning', 'Weeding']
          activities[0]

Out[36]: 'Tea'

Out[62]: [5, 10, 5, 10, 5, 10]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator. For example:

The farm subsidises food for the workers at a lower-than-market rate.

In response, we will update the list with 'cabbage' and 'milk':

```
In [41]: M products = ['Kales', 'Avocado', 'cabbage', 'spinach']
```



Notice the use of methods as introduced earlier in the next few cells of code. They all provide functionality that would not be obvious/easy to code up and let us work with lists in a more intuitive way.

A quick way to read about / learn what different methods do, is to click inside the brackets of a method, and press `[SHIFT]+[TAB]+[TAB]` to see an explanation of what it does:

```
In [43]: # Adding elements at the back

print("Initially before adding: ", products)

products.append("cabbage") # note the use of the Append() method, and the argument 'cabbage', and look what it does!
products.append("milk")

print("Finally: ", products)

Initially before adding: ['Kales', 'Avocado', 'cabbage', 'spinach']
Finally: ['Kales', 'Avocado', 'cabbage', 'spinach', 'cabbage', 'milk']
```

```
In [45]: # Deleting the last element

print("Initially before deleting last element: ", products)

products.pop()

print("Finally: ", products)

Initially before deleting last element: ['Kales', 'Avocado', 'cabbage', 'spinach', 'cabbage', 'milk']
Finally: ['Kales', 'Avocado', 'cabbage', 'spinach', 'cabbage']
```

```
In [47]: # Deleting any element

print("Initially before deleting any element: ", products)

del products[2]

print ("After deleting value at index 2 : ", products)

Initially before deleting any element: ['Kales', 'Avocado', 'cabbage', 'spinach', 'cabbage']
After deleting value at index 2 : ['Kales', 'Avocado', 'spinach', 'cabbage']
```

```
In [ ]: # Getting the Length of the List

length = len(list2)

print("Length of the list: ", length)
```

Other methods include:

`list.count('Avocado')`: returns count of how many times obj occurs in list

`list1.extend(list2)`: appends the contents of list2 to list1

`list.index('Avocado')`: returns the lowest index in list that obj appears

`list.insert(3, "Mangoes")`: inserts object obj into list at offset index

`list.remove("tea")`: removes object obj from list

`list.reverse()`: reverses objects of list in place



Tuples

Tuples are very similar to lists, in that you can do operations with them (+ and *), but the difference is that they are immutable. This means that once they are created, you can't change their individual elements!

Tuples are created with objects that are put in between normal brackets () as seen below:

```
In [38]: M (5,10,15,20) # this is a tuple!
Out[38]: (5, 10, 15, 20)

In [9]: M tuple_activity = ('Tea', 'Plucking', 'Pruning', 'Weeding')
In [10]: M tuple_product = ('Kales', 'Avocado', 'cabbage', 'spinach', 'milk')
In [11]: M print("tuple_activity[0]: {}".format(tuple_activity[0]))
           print ("tuple_product[1:3]: {}".format(tuple_product[1:3]))
           tuple_activity[0]: Tea
           tuple_product[1:3]: ('Avocado', 'cabbage')

In [37]: M added_tuple = tuple_activity + tuple_product
           print(added_tuple)
           ('Tea', 'Plucking', 'Pruning', 'Weeding', 'Kales', 'Avocado', 'cabbage', 'spinach', 'milk')
```

Dictionaries

A dictionary can also be any length, since it is a container, but here the values come in pairs rather than alone. The idea is that you can use the first value to look up the value of the second (just like in a normal dictionary!). The elements in a dictionary are called key-value pairs.

Dictionaries are created using curly brackets {}, a comma , to separate pairs of values, and a colon : to separate the first and second value of any key-value pair. To select elements in this dictionary, instead of using the index of the container, we use its key. This allows us to keep a record of information all in one place. If we go back to the amount of tax that we must pay for plucking tea, it is possible for us to keep all our values for the calculation in one place.



Other dictionary methods include:

`dict.clear()`: removes all elements of dictionary dict
`dict.copy()`: returns a shallow copy of dictionary dict
`dict.fromkeys()`: creates a new dictionary with keys from seq and values set to value
`dict.get("Price", default=None)`: for key, returns value or default if key is not in dictionary
`dict.has_key("price")`: removed, use the `in` operation instead
`dict.items()`: returns a list of dict's (key, value) tuple pairs
`dict.keys()`: returns a list of dictionary dict's keys
`dict.setdefault("Amount", default = None)`: similar to `get()`, but will set `dict[key]`: default if key is not already in dict
`dict.update(dict2)`: adds dictionary dict2's key-values' pairs to dict
`dict.values()`: returns list of dictionary dict's values

Now that we know all the basic types available in Python, we can start doing some useful things with them!

Let's say that we wanted to calculate our daily income using Farm Valleydrum prices, and all we had was a dictionary with the activity and prices of each activity:

```
In [31]: prices = {'tea_plucking_per_kilo' : 10,  
                  'tea_pruning_per_bush' : 7,  
                  'coffee_picking_per_kilo' : 7,  
                  'coffee_pruning_per_bush' : 12}
```

We know that to calculate our daily income, we would need to multiply each one of the prices by the number of kilos/ bushes that were picked. So how can we tell Python to do this?

The answer: with loops.




Loops

A powerful element of programming is that we can cycle through every single element in any container, and do something with it. The general structure of a 'for' loop looks like this:

```
In [ ]: M for element in container:
        do something here
```

What Python is doing here, is going through every element in a container, one by one, and executing the code that is written on the next line. Notice that the second line in the pseudo-code above does not start at the very left-hand side of the line, instead it is indented. This is part of the 'code' in Python, and is important, as it tells Python that everything indented below the statement `for element in container:` is part of the loop and should be run numerous times.

Indent  `for element in container:`
`do something here`

What is happening here, is that Python interprets all the code that is indented as being part of the for-loop. So, if we want to do something to an element, the code must be indented. You can create an indent by either pressing the 'tab' key or pressing spacebar four times.

Assume we have a person working for five days and the number of kilos he plucks is put in the container as Monday=20, Tuesday= 30, Wednesday= 55, Thursday= 20, Friday= 40. To work out his daily wage, we can use:

```
container = [20,30,55,20,40]
```

```
In [3]: M container = [20,30,55,20,40]
        for element in container:
            print(element*10)

200
300
550
200
400
```



We can use this functionality to calculate our total daily income:

If we organise the amount of tea plucked and the amount of weeding with their prices in the following way, we can use a loop to calculate the total income:

Amount, measured in kilograms (kg), for Monday is 20kg, Tuesday weeding, Wednesday weeding, Thursday is 20kg and Friday is 40kg.

```
In [6]: M amounts = [20,1,1,20,40]
        prices = [10,310,310,10,10]

In [9]: M total = 0 # we set the total to be 0 to start with and will add numbers to it as we go through the loop
        for i in range(5):
            total = total + (amounts[i] * prices[i])
        total

Out[9]: 1420
```

Note how the individual elements of the list can be manipulated as normal numbers and not lists!

We could also do the same calculation using the 'prices' dictionary, although it would look slightly different:

```
In [29]: M # define our variables

        prices_dict = {'tea_plucking_per_kilo' : 10,
                       'tea_pruning_per_bush' : 7,
                       'coffee_picking_per_kilo' : 7,
                       'coffee_pruning_per_bush' : 12}

        amounts = [45, 10, 20, 5]

        # set our total to be zero to start with
        total = 0

        # write our loop to calculate total amount earned in a day

        # the values i, activity, are variables which take on the values
        # of each element in the enumerate(prices_dict) object
        # the enumerate() function adds another variable, an index, to the loop

        for i, activity in enumerate(prices_dict):
            # add the amount multiplied by the price of each activity to the total
            total = total + (amounts[i] * prices_dict[activity])

        total

Out[29]: 720
```



Click the link below to view a programme which visualises what happens in every step!

<https://t1p.de/k627>

Click the 'next' button in the link to get the programme to read the next line of code.



3.0

Libraries



LIBRARIES (OTHERWISE KNOWN AS MODULES)

A library is a simple but very powerful concept. At its core, it is simply code which has already been written and can be imported so that we can use that functionality. There are different libraries to speed up coding time for almost any purpose. Popular libraries include:

- **Plotting libraries.** The original plotting library for Python is called Matplotlib and is frequently used to display plots such as: bar graphs, scatter plots, distribution plots and much more.
- **Mathematical libraries.** Examples include math, SciPy and NumPy. These libraries make long calculations much easier and are used extensively in the scientific community.
- **Data preparation libraries.** The most common package is Pandas, which stands for 'Python Data Analysis Library', and is used to access and process data.
- **Machine learning libraries.** Scikit-learn and Keras are libraries that are used to develop machine learning models and make various predictive insights.

These are just a few of the more common libraries used in Python, however a wider variety is available, and range from signal processing to even art-creation with Python!

There are two steps to getting a library to successfully work in your notebook, which includes:

- downloading the library

There are multiple ways to do this. Since we are in a notebook, the easiest way is to type the 'magic' command: `!pip install [LIBRARY HERE]` for example, `!pip install numpy`. Since you have already downloaded Anaconda, it comes with many of the more popular libraries pre-downloaded, so for something like NumPy or Pandas, this step can be skipped entirely. However, if you want a specific functionality then you will have to download it!

- importing the library

This is much simpler but **must** be done each time you open a new notebook. To import, simply type `import [LIBRARY HERE]` where the library can be called after this step. You can also write as `[ABBREVIATION]` which is common when you know you will have to type out the word a lot in the future. A common approach for importing Pandas is to type `import pandas as pd`.

```
In [27]: !pip install pandas

Requirement already satisfied: pandas in c:\users\sergeibetishchev\appdata\roaming\python\python37\site-packages (0.25.3)
Requirement already satisfied: python-dateutil>=2.6.1 in c:\programdata\anaconda3\lib\site-packages (from pandas) (2.8.1)
Requirement already satisfied: numpy>=1.13.3 in c:\programdata\anaconda3\lib\site-packages (from pandas) (1.18.1)
Requirement already satisfied: pytz>=2017.2 in c:\programdata\anaconda3\lib\site-packages (from pandas) (2019.3)
Requirement already satisfied: six>=1.5 in c:\programdata\anaconda3\lib\site-packages (from python-dateutil>=2.6.1->pandas) (1.14.0)
```

We see from the output that Pandas is already installed in Anaconda:

```
In [28]: ! # import pandas Library
import pandas as pd
```

Data Acquisition

There are various formats for a dataset, `.csv`, `.JSON`, `.xlsx` etc. The dataset can be stored in different places, either on your local machine or occasionally online. In this section, you will learn how to load a dataset into our Jupyter Notebook. In our case, we will read in data for Mapera Farm, which is in an Excel (`.xlsx`) format. Let's use this dataset as an example to practice data reading.

The Pandas Library is a useful tool that enables us to read various datasets into a dataframe. Our Jupyter notebook platforms have a built-in Pandas Library so that all we need to do is import Pandas without installing.

Read Data

We use the `pandas.read_csv()` method to read the `.csv` file. In the brackets, we put the file path along with quotation marks, so that Pandas will read the file into a dataframe from that address. The file path can either be a URL or your local file address. Since the data does not include headers, we can add an argument `header = None` inside the `read_excel()` method, so that Pandas will not automatically set the first row as a header. You can also assign the dataset to any variable you create:

```
In [56]: ! # import pandas Library
import pandas as pd

# Read data from an excel file
df = pd.read_csv('mapera_farm.csv', index_col=0)
```



The Power of Methods

After reading the dataset, we can use the `dataframe.head(n)` method to check the top ‘n’ rows of the dataframe; where ‘n’ is an integer. Contrary to `dataframe.head(n)`, `dataframe.tail(n)` will show you the bottom ‘n’ rows of the dataframe. This is as simple as calling the `.head()` method, as all the work to display the rows in our notebook has been completely removed:

```
In [57]: # show the first 5 rows using dataframe.head() method
print("The first 5 rows of the dataframe")
df.head(5)
```

The first 5 rows of the dataframe

```
Out[57]:
```

	dd	kg	dd.1	kg.1	dd.2	kg.2	dd.3	kg.3	dd.4	kg.4	...	end	other	ded	ded.1	other.1	Unnamed: 77	Unnamed: 78	Unnamed: 79	Unnam
	NaN	1-Jan	NaN	2-Jan	NaN	3-Jan	NaN	4-Jan	NaN	5-Jan	NaN	...	net	NaN	adv	end	NaN	NaN	NaN	N
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	5,000	N
CHRIS AUKA	NaN	NaN	NaN	1	NaN	NaN	20	NaN	18	1	NaN	...	4,040	NaN	NaN	180	NaN	5,170	NaN	N
CHRISTOPHER ONGARI	NaN	NaN	NaN	51	NaN	58	NaN	NaN	1	NaN	...	6,560	NaN	NaN	NaN	NaN	6,910	NaN	NaN	N
DAVID MGAKA	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	6,970	NaN	NaN	440	NaN	7,410	NaN	NaN	N

5 rows × 81 columns

Save Dataset

Correspondingly, Pandas enables us to save the dataset to `.csv` by using the `dataframe.to_csv()` method. You can also add the file path and name along with quotation marks inside the brackets.

For example, if you were to save the dataframe ‘df’ as ‘`automobile.csv`’ to your local machine, you may use the syntax below:

```
In [ ]: df.to_csv("automobile.csv", index=False)
```

Read/Save Other Data Formats

Data Format	Read	Save
csv	<code>pd.read_csv()</code>	<code>df.to_csv()</code>
json	<code>pd.read_json()</code>	<code>df.to_json()</code>
excel	<code>pd.read_excel()</code>	<code>df.to_excel()</code>
hdf	<code>pd.read_hdf()</code>	<code>df.to_hdf()</code>
sql	<code>pd.read_sql()</code>	<code>df.to_sql()</code>
...



Basic Insight of Dataset

After reading data into the Pandas dataframe, it is time for us to explore the dataset. There are several ways to obtain essential insights of the data to help us better understand our dataset.

Data Types in Pandas

Data has a variety of types. The main types stored in Pandas' DataFrames are object, float, int, bool and datetime64. In order to learn about each attribute, it is useful for us to know the data type of each column in Pandas:

```
In [58]: df.dtypes
Out[58]: dd          object
kg          float64
dd.1        object
kg.1        object
dd.2        object
...
Unnamed: 77 object
Unnamed: 78 float64
Unnamed: 79 object
Unnamed: 80 float64
Unnamed: 81 float64
Length: 81, dtype: object
```

Describe

If we would like to get a statistical summary of each column, such as: count, column mean value, and column standard deviation, we use the `'describe'` method:

```
In [60]: df.describe()
Out[60]:
```

	kg	kg.4	kg.5	kg.11	kg.17	kg.18	kg.25	other	other.1	Unnamed: 78	Unnamed: 80	Unnamed: 81
count	1.0	1.0	27.000000	1.0	24.000000	1.0	1.0	4.0	0.0	0.0	0.0	0.0
mean	0.0	0.0	63.407407	0.0	70.916667	0.0	0.0	437.5	NaN	NaN	NaN	NaN
std	NaN	NaN	159.076242	NaN	167.029265	NaN	NaN	125.0	NaN	NaN	NaN	NaN
min	0.0	0.0	13.000000	0.0	15.000000	0.0	0.0	250.0	NaN	NaN	NaN	NaN
25%	0.0	0.0	20.500000	0.0	23.000000	0.0	0.0	437.5	NaN	NaN	NaN	NaN
50%	0.0	0.0	29.000000	0.0	32.000000	0.0	0.0	500.0	NaN	NaN	NaN	NaN
75%	0.0	0.0	46.500000	0.0	51.000000	0.0	0.0	500.0	NaN	NaN	NaN	NaN
max	0.0	0.0	856.000000	0.0	851.000000	0.0	0.0	500.0	NaN	NaN	NaN	NaN

This method will provide various summary statistics, excluding NaN (Not a Number) values.

This shows the statistical summary of all numeric-type (`int`, `float`) columns.

However, what if we were to check all the columns, including those that are of type object?:

```
In [61]: # describe all the columns in "df"
df.describe(include = "all")
```

Out[61]:

	dd	kg	dd.1	kg.1	dd.2	kg.2	dd.3	kg.3	dd.4	kg.4	...	end	other	ded	ded.1	other.1	Unnamed: 77	Unnamed: 78	Unnamed: 79	Unnamed: 80	Unr
count	5	1.0	22	23	18	29	16	30	26	1.0	...	52	4.0	22	25	0.0	50	0.0	1	0.0	
unique	3	NaN	3	21	3	25	3	24	3	NaN	...	50	NaN	16	23	NaN	45	NaN	1	NaN	
top	1	NaN	1	48	1	76	1	25	1	NaN	...	5,340	NaN	100	800	NaN	3,360	NaN	5,000	NaN	
freq	3	NaN	20	2	16	2	14	2	24	NaN	...	2	NaN	3	2	NaN	3	NaN	1	NaN	
mean	NaN	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	...	NaN	437.5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
std	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	125.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
min	NaN	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	...	NaN	250.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
25%	NaN	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	...	NaN	437.5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
50%	NaN	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	...	NaN	500.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
75%	NaN	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	...	NaN	500.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
max	NaN	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	...	NaN	500.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

11 rows × 81 columns

Now, it provides the statistical summary of all the columns, including object-typed attributes. We can now see how many unique values, which is the top value and the frequency of top value in the object-typed columns. Some values in the table above show as 'NaN' because those numbers are not available as a particular column type.



4.0

Functions

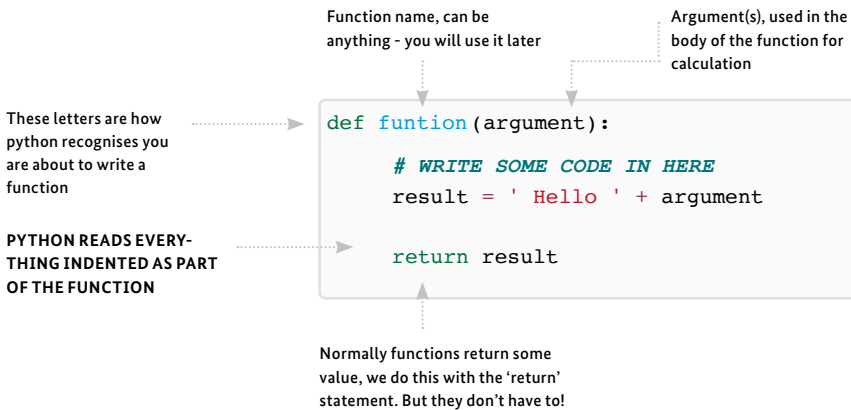


FUNCTIONS

Functions are conceptually similar and act as a piece of functionality that can be called multiple times, so that you don't have to write the code every time. The syntax looks like this:

```
In [205]: def function(argument):  
          # WRITE SOME CODE IN HERE  
          result = 'Hello ' + argument  
          return result
```

This is perhaps one of the simplest functions where the important features of this bit of code are as follows:



Are Functions and Methods the Same Thing?

Almost, but no.

You may have noticed the similarity between the function we just built, and a 'method' defined in the previous workshop. The two are almost the same since they do indeed have the same syntax when comparing functions with built-in methods. The difference is subtle, and even though they are often referred to in the same way, they do serve slightly different purposes:

- a Function is something that is written as a standalone piece of functionality
- a Method is a function which is attached to an object

This means that if you define a function somewhere in your code, you can call it anywhere and without writing out the object first, i.e. `Function(arguments)`, while a method has to be called on the object, using the ``.`` symbol to state that you are going to call a method of an object.

The syntax for this is:

`Object.method(arguments)`, and the only cases you see the syntax of `method(arguments)` are for Python's special built-in functions.





5.0

The DataFrame



THE DATAFRAME

According to the Pandas' API documentation (where you can also read more about what the library can do for you), a DataFrame is:

A 2-dimensional labelled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most used Pandas object. Like Series, DataFrame accepts many kinds of input:

- dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- structured or record ndarray
- a Series (another Pandas object, it is a single column of a Pandas DataFrame)
- another DataFrame

Pandas Series

Series is a one-dimensional labelled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). A Pandas series can hold many different types of data, including:

- a Python dictionary
- an array (or list)
- a scalar value (like 5)

To read more about the different objects in the Pandas libraries, you can visit the website here:

https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html

In this playbook we will be focusing on the practical application of Pandas rather than repeat the information that is available above. Concepts will be introduced as and when they are required, but keep in mind that there is a full set of information describing the library in the link!



Head and Tail

To view a small sample of a Series or DataFrame object, use the `'head()'` and `'tail()'` methods. The default number of elements to display is five, but you may pass a custom number.

It's useful to run these two methods straight after you've read in the data. This is to make sure that the columns are as you expect, and that the read-in method that was called has given you the data that you need to work with.

In [253]: `df.head()`

Out[253]:

	Date	CHRIS AUKA	CHRISTOPHER ONGARI	DAVID MGAKA	DAVID NGO	DAVID AOGA	DELVIN KIMBOI	DENNIS OMBARI	DICKSON KEBAKA	ELAISHA OARU	...	RISPER SIRO	ROBERT AIYEKE
0	01-Jan	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN
1	02-Jan	NaN	51.0	NaN	NaN	NaN	40.0	8.0	NaN	63.0	...	NaN	61
2	03-Jan	20.0	58.0	NaN	28.0	25.0	43.0	48.0	NaN	76.0	...	NaN	91
3	04-Jan	18.0	NaN	NaN	32.0	25.0	41.0	27.0	NaN	48.0	...	NaN	77
4	05-Jan	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN

5 rows x 51 columns

Dealing with Missing Data

If you have called the `read_csv` method, any values that were originally empty in the `.csv` will be converted to `'NaN'` values. These are generally difficult to work with, as they will hinder functionality if left unchecked (such as plotting or if you want to make a function apply to each cell in a column; that function may break if it sees an `'NaN'` value).



For a more comprehensive explanation of working with missing data, it is possible to visit the link below to see how the creators of Pandas envisioned dealing with missing data:

https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html#missing-data

In most cases, it will be that we want to replace the `'NaN'` values with something which is friendlier to work with. Since we know that our data is numeric, a common option is to replace the `'NaN'` values with 0. This makes sense in our case, since workers with `'NaN'` values in any cell would not have picked any tea on that day.

```
In [262]: # we can use the .fillna() method to replace NaN values.
# The 'inplace' argument means that the object df will be changed
df.fillna(value = 0, inplace = True)
```

```
In [263]: # now we can check our dataframe again to see if the changes we have made
# are reflected in our new dataframe object
df.head()
```

Out[263]:

	Date	CHRIS AUKA	CHRISTOPHER ONGARI	DAVID MGAKA	DAVID NGO	DAVID AOGA	DELVIN KIMBOI	DENNIS OMBARI	DICKSON KEBAKA	ELAISHA OARU	...	RISPER SIRO	ROBERT AIYEKE	
0	01-Jan	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0	
1	02-Jan	0.0	51.0	0.0	0.0	0.0	40.0	8.0	0.0	63.0	...	0.0	61	
2	03-Jan	20.0	58.0	0.0	28.0	25.0	43.0	48.0	0.0	76.0	...	0.0	91	
3	04-Jan	18.0	0.0	0.0	32.0	25.0	41.0	27.0	0.0	48.0	...	0.0	77	
4	05-Jan	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0	

5 rows × 51 columns

It worked!

We could have decided to fill our values with something related to the same column/row in which they were found, which is what the `'method'` argument is for (remember, press `[SHIFT]+[TAB]+[TAB]` to view the details of a method).



For example, we could have chosen to fill the values with previous/future values seen in the column/row by using the keyword `'backfill'`; this would turn the NaN values into the same value as seen later on in the column. Of course, since we just converted all our `'NaN'` values to zeroes, this function will no longer do anything, and just displays what we already have.

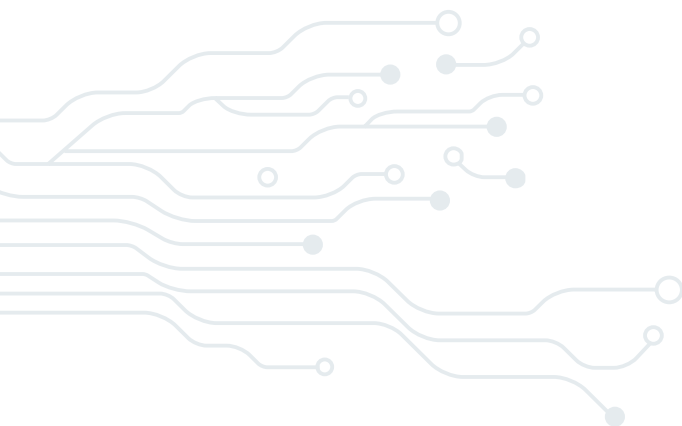
```
In [264]: df.fillna(method = 'backfill').head()
```

Out[264]:

	Date	CHRIS AUKA	CHRISTOPHER ONGARI	DAVID MGAKA	DAVID NGO	DAVID AOGA	DELVIN KIMBOI	DENNIS OMBARI	DICKSON KEBAKA	ELAISHA OARU	...	RISPER SIRO	ROBERT AIYEKE
0	01-Jan	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0
1	02-Jan	0.0	51.0	0.0	0.0	0.0	40.0	8.0	0.0	63.0	...	0.0	61
2	03-Jan	20.0	58.0	0.0	28.0	25.0	43.0	48.0	0.0	76.0	...	0.0	91
3	04-Jan	18.0	0.0	0.0	32.0	25.0	41.0	27.0	0.0	48.0	...	0.0	77
4	05-Jan	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0

5 rows x 51 columns

< >



Data Cleaning - Worked Example

We can quickly check to see what the 'types' of the individual columns are, by using the attribute `'object.dtypes'`:

```
In [42]: df.dtypes
```

Out[42]:	Date	object
	CHRIS AUKA	float64
	CHRISTOPHER ONGARI	float64
	DAVID MGAKA	float64
	DAVID NGO	float64
	DAVID AOGA	float64
	DELVIN KIMBOI	float64
	DENNIS OMBARI	float64
	DICKSON KEBAKA	float64
	ELAISHA OARU	float64
	ELIZABETH MACHUMARI	float64
	FLORENCE NJOKI	float64
	FLORENCE NYABIRO	float64
	GEOFFREY AYIEKA	float64
	GEORGE MORAA	float64
	GLADYS KWAMBOKA MOSE	float64
	GRACE WAIRIMU	float64
	HANNAH WAITHERA NGUGI	float64
	HELLEN OYALO	float64

The process of 'data cleaning' just means asking the question: 'Does the data look like what we were expecting?'

Everything looks correct, except for the column under `'ROBERT AIYEKE'`, which shows that the column type is an object. Why is this the case? We were expecting numbers in there...

```
In [265]: # Let's have a closer look:
list(df['ROBERT AIYEKE'])
```

Out[265]:	[0,
	'61',
	'91',
	'77',
	0,
	'63',
	'106',
	'162',
	'161',
	'73',
	'59',
	0,
	'183',
	'193',
	'71',
	'66',
	'53',
	'20',
	0,



It appears that this row is a mix of zeroes and numbers but stored as a string (we can tell by the quotation marks around the numbers).

```
In [279]: # Let's see if we can convert this column to a float like the other columns:
```

```
try:
    df['ROBERT AIYEKE'].astype(float)
except:
    print("you can't do this, it breaks!")
```

```
you can't do this. it breaks!
```

For some reason, Python isn't letting us convert this column into numbers, even though the strings look like they contain numbers.

On closer inspection, there's one entry that Python doesn't know what to do with. Can you spot it?

```
In [282]: df['ROBERT AIYEKE'][80:90]
```

```
Out[282]: 80    0
          81    0
          82    0
          83    0
          84    0
          85
          86    0
          87    0
          88    0
          89    0
          Name: ROBERT AIYEKE, dtype: object
```

There isn't supposed to be an empty cell! What is it?

```
In [285]: df['ROBERT AIYEKE'][85]
```

```
Out[285]: 0.0
```



It seems that it is a string containing spaces. We can remove whitespace with the method `.strip()`, which belongs to the string type. Even though this method is not directly available to a Pandas Series, they have objects called `_accessors_` which let you use functionality from Python's base types on their strings. The string accessor can be accessed via the command `'.str'`.

```
In [284]: # Let's first convert everything to a string so that everything in the series
# is being processed in the same way

df['ROBERT AIYEKE'] = df['ROBERT AIYEKE'].astype(str)

# now apply the accessor and use the strip() method
df['ROBERT AIYEKE'] = df['ROBERT AIYEKE'].str.strip()

# now that we have some empty (string) values in some cells we need to do something with these:
# one idea would be to replace empty string values with the value 0,
# so that it can be converted into a number
df.loc[df['ROBERT AIYEKE'] == '', 'ROBERT AIYEKE'] = '0'

# finally we can convert the column to a float type
df['ROBERT AIYEKE'] = df['ROBERT AIYEKE'].astype(float)

print('Done!')
```

Done!



6.0

Indexing and Selecting Data in Pandas



INDEXING AND SELECTING DATA IN PANDAS

This section is all about how to access specific cells or groups of cells in a dataframe. It is a key part of data manipulation in Pandas and is core to being able to work with data in Python.

There are three main ways in which to select data from a dataframe:

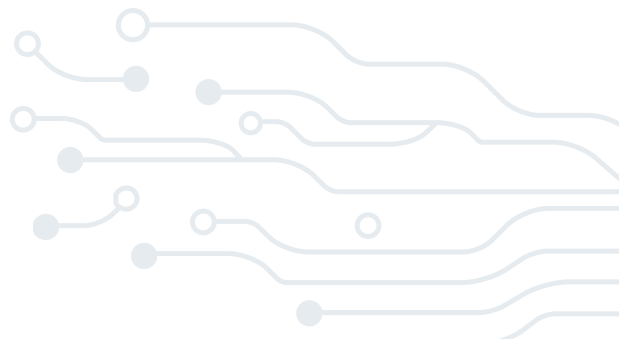
[]

The simplest way to select anything from a dataframe is by selecting `dataframe['column_name']` which returns the entire column (known in the Pandas library as a 'Series')

.loc

This is primarily **label** based but may also be used with a boolean array. `.loc` will raise a `KeyError` when the items are not found. Inputs that are allowed include:

- a single label, e.g. 5 or 'a' (note that 5 is interpreted as an index label – this use is not an integer position along the index)
- a list or array of labels ['a', 'b', 'c']
- a slice object with labels 'a':'f' (note that contrary to usual Python slices, both the start and the stop are included when present in the index – see 'Slicing with labels and Endpoints are inclusive')
- a Boolean array (any NA values will be treated as 'False')
- a callable function with one argument (the calling Series or DataFrame) that returns valid outputs for indexing (one of the above)



.iloc

This is primarily integer position based (from 0 to length-1 of the axis) but may also be used with a boolean array. `.iloc` will raise an `IndexError` if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing (this conforms with Python/NumPy slice semantics). Allowed inputs are:

- an integer e.g. 5
- a list or array of integers, e.g. [4, 3, 0]
- a slice object with ints, e.g. 1:7
- a boolean array (any NA values will be treated as False)
- a callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

Worked Examples

```
In [16]: M # Let's try selecting some of our data and check its type
```

```
print(df['Date'])
type(df['Date'])
```

```
0      01-Jan
1      02-Jan
2      03-Jan
3      04-Jan
4      05-Jan
...
208    27-Jul
209    28-Jul
210    29-Jul
211    30-Jul
212    31-Jul
Name: Date, Length: 213, dtype: object
```

```
Out[16]: pandas.core.series.Series
```

A useful variable to know in dataframes is the `.columns` variable. This is not a method, which we can see since there are no brackets after the name. It lists all the columns' names in our dataframe, which is especially helpful when the list is too long to see using the `'.head()'` method!

In [17]: `df.columns`

```
Out[17]: Index(['Date', 'CHRIS AUKA ', 'CHRISTOPHER ONGARI ', 'DAVID MGAKA ',
'DAVID NGO', 'DAVID AOGA', 'DELVIN KIMBOI', 'DENNIS OMBARI',
'DICKSON KEBAKA ', 'ELAISHA OARU', 'ELIZABETH MACHUMARI',
'FLORENCE NJOKI', 'FLORENCE NYABIRO', 'GEOFFREY AYIEKA',
'GEORGE MORAA', 'GLADYS KWAMBOKA MOSE', 'GRACE WAIRIMU',
'HANNAH WAITHERA NGUGI', 'HELLEN OYALO ', 'HARRISON MATEKE',
'JOHN OBARE', 'JOEL BIKO ', 'JAMES OMBATI', 'JOSEPH KHAMAKAYA',
'JOSEPHINE BOCHABERI', 'JOSEPHINE KERUBO ', 'JOSEPHINE NYAGITARI',
'JOYCE THUO', 'KENNEDY NYAKWAKA', 'MARGARET ATHIAMBO',
'MARGARET ORIKA', 'MARGARET GACHAGWA', 'MARK WEKESA NAMASAKA',
'MELLEN MUTHONI', 'MONICA KIVETI', 'MOSES NYARABIRO', 'PACIFICA OTEKA',
'PERIS MACHACHIRE', 'PHILLIS APIYO', 'RAPHAEL OTETE',
'REBECCA NYASIAKI', 'RISPER SIRO', 'ROBERT AIYEKE', 'RUTH NYAKATCH',
'SABET KAPAKE', 'SAMSON NGOVI', 'SAMUEL MATANO', 'SAM NTABIRO',
'WALTER WANYONYI ', 'YUNESI KERUBO NYANUMBA', 'ZEBLON NYALENDE'],
dtype='object')
```

In [313]: `# if we apply the same method on a series, we get back a singular value`

```
s = df['Date']
print(s, '\n')
print('s[3] output: {}'.format(s[3]))
df['Date'][3]
0      01-Jan
1      02-Jan
2      03-Jan
3      04-Jan
4      05-Jan
...
208    27-Jul
209    28-Jul
210    29-Jul
211    30-Jul
212    31-Jul
Name: Date, Length: 213, dtype: object

s[3] output: 04-Jan
Out[313]: '04-Jan'
```

In general, we can select rows and columns in a dataframe using the following syntax:

```
DataFrame.loc[row labels , column labels]
```

What if we wanted to do something like select all the days on the first of the month of the dataframe?



In [104]: `# Select first day of each month in the dataframe`

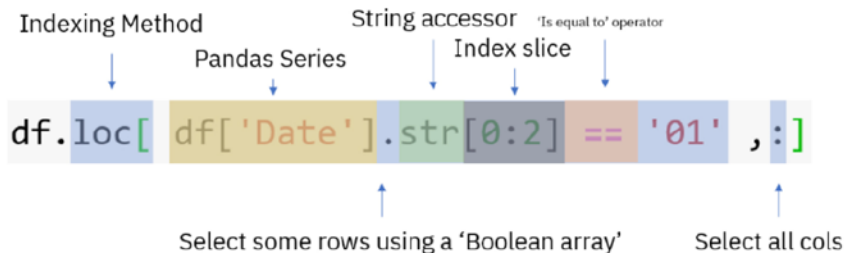
```
df.loc[ df['Date'].str[0:2] == '01' ,:]
```

Out[104]:

	Date	CHRIS AUKA	CHRISTOPHER ONGARI	DAVID MGAKA	DAVID NGO	DAVID AOGA	DELVIN KIMBOI	DENNIS OMBARI	DICKSON KEBAKA	ELAISHA OARU	...	RISPER SIRO	ROBERT AIYEKE
0	01-Jan	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
31	01-Feb	27.0	0.0	40.0	24.0	30.0	0.0	29.0	0.0	0.0	...	0.0	0.0
60	01-Mar	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
91	01-Apr	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	30.0	...	72.0	0.0
121	01-May	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
152	01-Jun	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
182	01-Jul	0.0	19.0	31.0	36.0	48.0	0.0	26.0	48.0	0.0	...	36.0	33.0

7 rows × 51 columns

Let's understand what is happening in this index:



In essence, the first statement before the comma `','` is what selects the correct rows of the dataframe, where the second statement is the default character to `'select all'`.

The statement `df['Date'].str[0:2] == '01'` returns a list of True/False values, and is what is removing certain rows from the dataframe `df`, which we can see below:

```
In [20]: M df['Date'].str[0:2] == '01'
Out[20]: 0      True
         1      False
         2      False
         3      False
         4      False
         ...
        208     False
        209     False
        210     False
        211     False
        212     False
         Name: Date, Length: 213, dtype: bool
```

```
In [ ]: M DataFrame.loc[ INDEXER , :]
```

Using the `df.iloc` function is, in general, simpler, since the data that you wish to select, corresponds with the values that go inside the row and column's number.

So, for example, to select the cell in the first row and column in the dataframe, we would use the following command:

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

```
In [21]: M # Doing this selection returns just the most 'top-left' value in the dataframe
         df.iloc[0,0]
Out[21]: '01-Jan'
```



Armed with this basic knowledge, we can now figure out relatively complex calculations such as understanding how much tea one person plucked on the first day of each month in total, by first selecting the relevant cells and then summing them all:

```
In [314]: # total number of tea picked on the 1st of every month
total_on_1st = df.loc[df['Date'].str[0:2] == '01', 'CHRIS AUKA '].sum()
print(total_on_1st)
27.0
```

...or look at the total amount of tea picked on each day:

```
In [316]: df.iloc[:,1:].sum(axis = 1)
# notice the use of the .sum() method to make our lives easier in adding up all the cells!

Out[316]: 0      0.0
1    1272.0
2    1658.0
3    1272.0
4      0.0
...
208    641.0
209    895.0
210    796.0
211    857.0
212      0.0
Length: 213, dtype: float64
```

Note that since the list is long, Pandas automatically ‘shortens’ the output with the three dots ‘...’ in between the beginning and end of the object. There are numerous ways to get around this.



One is to convert the object to a list using the `'list()'` built-in method. Another is to change how much data Pandas displays by default. This can be done in the following way:

```
In [113]: M data_to_view = df.iloc[:,1:].sum(axis = 1)
          with pd.option_context('display.max_rows', None, 'display.max_columns', None):
              display(data_to_view)
```

0	0.0
1	1272.0
2	1658.0
3	1272.0
4	0.0
5	856.0
6	1888.0
7	2253.0
8	2502.0
9	2413.0
10	1815.0
11	0.0
12	2399.0
13	3211.0
14	2379.0
15	1887.0
16	1731.0
17	851.0
18	0.0

...or look at the total amount of tea picked by each worker:

```
In [25]: M df.iloc[:,1:].sum(axis = 0)
```

```
Out[25]: CHRIS AUKA      3039.0
CHRISTOPHER ONGARI     4525.0
DAVID MGAKA            3029.0
DAVID NGO              4100.0
DAVID AOGA             4119.0
DELVIN KIMBOI          4362.0
DENNIS OMBARI          4405.0
DICKSON KEBAKA         4832.0
ELAISHA OARU           5221.0
ELIZABETH MACHUMARI     2879.0
FLORENCE NJOKI         4837.0
FRORENCE NYABIRO       4909.0
GEOFFREY AYIEKA        3538.0
GEORGE MORAA           3681.0
GLADYS KWAMBOKA MOSE   5917.0
GRACE WAIRIMU          4054.0
HANNAH WAITHERA NGUGI  7981.0
HELLEN OYALO           4155.0
HARRISON MATEKE        6780.0
```

It is possible to identify patterns in the data using clever indexing alone.

An even stronger combination is when it is paired with methods to aggregate data and plotting functionality, that is also built into Pandas.



7.0

Aggregation



AGGREGATION

The previous cell contained a way to aggregate the data using the method available to DataFrame. Other standard aggregation functions include:

`count()`: returns a count of all the values

`mean()`: returns the average value

`std()`: retruns the standard deviation

`min()`: retruns the minimum value

`median()`: returns the middle value

`max()`: retruns the largest value

In general, you can see all the methods available to the DataFrame object by typing `df.` and then pressing `[TAB]` afterwards. To see what the function does, select it, then press `[SHIFT]+[TAB]+[TAB]` inside the brackets `()` of the method for an explanation.

A very useful method is the `.apply()` method. This allows you to apply `_functions` that you created yourself_ to the entire dataset.

For example, let's say that we want to sum all the amounts of tea for each day, but only the values where the amount picked was larger than 30kg.



To accomplish this, we can use a function that takes an individual row and sums it, on the condition that the value is above 30, then apply this function to each row in the dataframe:

```
In [115]: def cond_sum(row):  
    # make sure the type is an integer  
    row = row.astype(int)  
  
    # this is an example of boolean indexing on a series and summing the resulting series  
    result = row[ row > 30 ].sum()  
  
    return result  
  
In [118]: # now if we run this function with a Series as the input we'll get a sensible output out:  
    my_series = df.iloc[1,1:]  
    cond_sum(my_series)  
  
Out[118]: 1132
```



Now we can apply this function to the entire dataframe in one line without having to do a loop using `'apply()'`. Don't be intimidated by the code `'lambda x:'` at the beginning of the apply arguments. This just means 'take every row and put it through my function'.

```
In [319]: df.iloc[:,1:].apply(lambda x: cond_sum(x))
```

```
Out[319]:
```

CHRIS AUKA	1591
CHRISTOPHER ONGARI	3589
DAVID MGAKA	2003
DAVID NGO	2547
DAVID AOGA	3516
DELVIN KIMBOI	4031
DENNIS OMBARI	3662
DICKSON KEBAKA	4415
ELAISHA OARU	4887
ELIZABETH MACHUMARI	2672
FLORENCE NJOKI	4033
FLORENCE NYABIRO	4294
GEOFFREY AYIEKA	3314
GEORGE MORAA	3136
GLADYS KWAMBOKA MOSE	5551
GRACE WAIRIMU	3412
HANNAH WAITHERA NGUGI	7012
HELLEN OYALO	3411
HARRISON MATEKE	6478
SAMUEL OROGO	2000

Success!



8.0

Plotting



PLOTTING

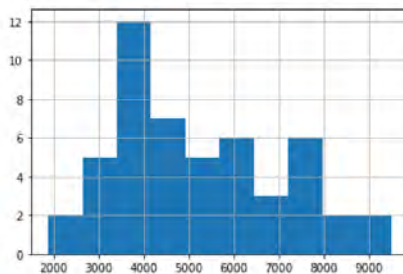
Basic - Series Plots

Once data has been selected and aggregated properly, there are many ways to plot something and often it is as simple as calling one method!

Historically, the most popular library is called 'Matplotlib', though there are new competitors coming up such as 'seaborn'. But for the most basic plots, Pandas has its own plotting methods in order to quickly visualise data.

To start, let us visualise the total amount of tea plucked by the workers with a histogram:

```
In [322]: my_data = df.iloc[:,1:].sum(axis = 0)
          my_data.hist(bins = 10)
          'This is already interesting, as from a visual inspection 4000kg seems to be anomalously high!'
Out[322]: 'This is already interesting, as from a visual inspection 4000kg seems to be anomalously high!'
```



To create a graph, it was as simple as calling the 'hist()' method which could be found by pressing `[SHIFT]+[TAB]`, and by pressing `[SHIFT]+[TAB]+[TAB]` inside the brackets of the method, where it is possible to find the 'bins' parameter and use it to change the width of each bar.

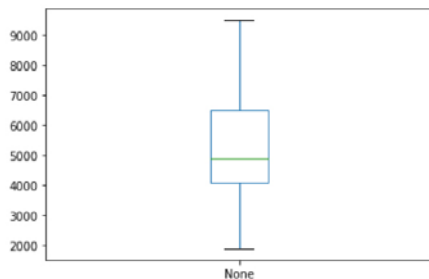
Other out-of-the-box data plot methods from the Pandas library includes `.plot()` which is a method for both Series and DataFrames and contains the following plots:

- `'line'`: line plot (default)
- `'bar'`: vertical bar plot
- `'barh'`: horizontal bar plot
- `'hist'`: histogram
- `'box'`: boxplot
- `'kde'`: kernel density estimation plot
- `'density'`: same as 'kde'
- `'area'`: area plot
- `'pie'`: pie plot
- `'scatter'`: scatter plot
- `'hexbin'`: hexbin plot

Some examples are shown below:

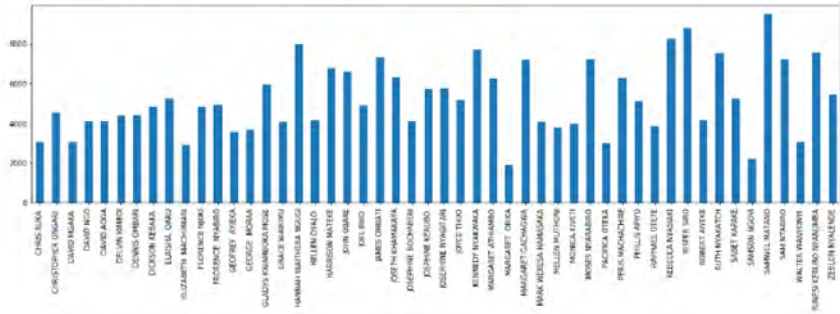
```
In [33]: my_data.plot(kind = 'box')
```

```
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x1e562752a08>
```



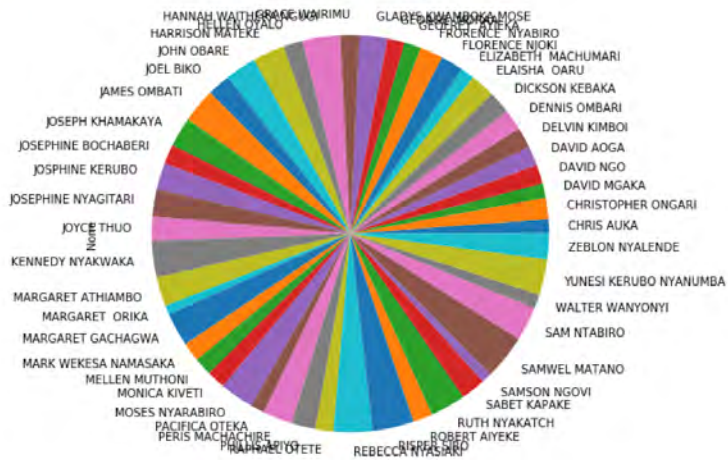
```
In [41]: # The figsize was made wider using the 'figsize' argument to read all the names
my_data.plot(kind = 'bar', figsize = (20,5))
```

```
Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x1e5663eae08>
```



```
In [46]: # It seems, as usual, pie charts are not a good way to display information
my_data.plot(kind = 'pie', figsize = (8,8))
```

Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x1e566eea288>



Dataframe Plots

What if we wanted to visualise the data of the entire dataframe in some way? We can also do this using the `.plot()` method.

Let's start by converting the 'Date' column to a 'datetime' type, so that Pandas can understand and plot it:

```
In [324]: # this command converts our date to a datetime type,
# now we can set it as the index and plot with it!

df['Date'] = pd.to_datetime(df['Date'] + '-2020', format = "%d-%b-%Y")

In [325]: # this command sets the date as the index, drops it
# and changes the original object 'df' so we don't have to reassign

df.set_index(df['Date'], inplace = True)

df.drop(labels = 'Date', axis = 1, inplace = True)

df.head()
```

Out[325]:

Date	CHRIS AUKA	CHRISTOPHER ONGARI	DAVID MGAKA	DAVID NGO	DAVID AOGA	DELVIN KIMBOI	DENNIS OMBARI	DICKSON KEBAKA	ELAISHA OARU	ELIZABETH MACHUMARI	...	RISPER SIRO
2020-01-01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0
2020-01-02	0.0	51.0	0.0	0.0	0.0	40.0	8.0	0.0	63.0	67.0	...	0.0
2020-01-03	20.0	58.0	0.0	28.0	25.0	43.0	48.0	0.0	76.0	49.0	...	0.0
2020-01-04	18.0	0.0	0.0	32.0	25.0	41.0	27.0	0.0	48.0	50.0	...	0.0
2020-01-05	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0

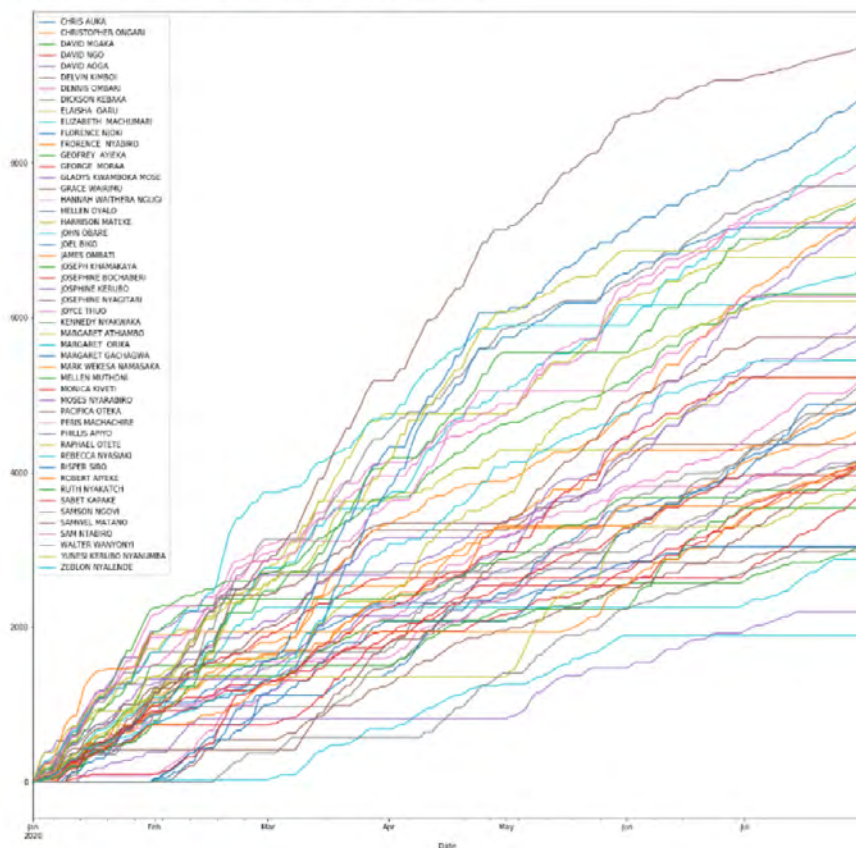
5 rows x 50 columns

Now we can plot this dataframe. Let's plot a time series, displaying each column as a different line (see next page):



```
In [98]: df.cumsum().plot(kind = 'line', figsize = (20,20))
```

```
Out[98]: <matplotlib.axes._subplots.AxesSubplot at 0x1e568856d88>
```



This is interesting, but hard to read since there are so many workers. We can try to analyse the correlation between each column and pick the ones with the least correlation in order to find the 'interesting' workers:

```
In [92]: # create a matrix of correlations and remove anything with a correlation above 0.75
# with a lambda function

cumulative = df.cumsum()
matrix = cumulative.corr()

def above_9(series):
    return series[series < 0.75]

# remove the entire row/column if it is full of na values and we are left with 8,
# different workers

display(matrix.apply(lambda x: above_9(x)).dropna(how = 'all', axis = 1). \
        dropna(how = 'all', axis = 0))
```

	DICKSON KEBAKA	ELIZABETH MACHUMARI	MARGARET ORIKA	MOSES NYARABIRO	PACIFICA OTEKA	RAPHAEL OTETE	ROBERT AIYEKE	WALTER WANYONYI
DICKSON KEBAKA	NaN	0.716267	NaN	NaN	NaN	NaN	NaN	NaN
ELIZABETH MACHUMARI	0.716267	NaN	0.696975	0.697911	0.742647	0.685462	0.743930	0.689926
MARGARET ORIKA	NaN	0.696975	NaN	NaN	NaN	NaN	NaN	NaN
MOSES NYARABIRO	NaN	0.697911	NaN	NaN	NaN	NaN	NaN	NaN
PACIFICA OTEKA	NaN	0.742647	NaN	NaN	NaN	NaN	0.718115	NaN
RAPHAEL OTETE	NaN	0.685462	NaN	NaN	NaN	NaN	NaN	NaN
ROBERT AIYEKE	NaN	0.743930	NaN	NaN	0.718115	NaN	NaN	NaN
WALTER WANYONYI	NaN	0.689926	NaN	NaN	NaN	NaN	NaN	NaN

We can already see that Elizabeth Machumari has a low correlation with many other workers. We can therefore conclude her tea picking is very different to the rest of the workers. Let's plot to see how!



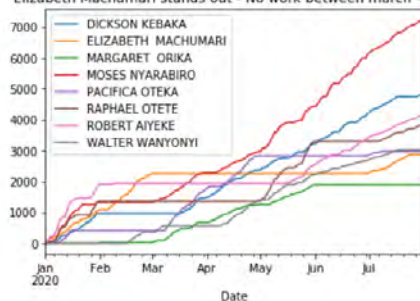
```
In [100]: # take the names of these workers and use them to filter the dataframe for plotting
```

```
workers = list(matrix.apply(lambda x: above_9(x)).dropna(how = 'all', axis = 1). \
                  dropna(how = 'all', axis = 0).index)
```

```
df.loc[:, df.columns.isin(workers)].cumsum(). \
    plot(title = 'Elizabeth Machumari stands out - No work between march - July!')
```

```
Out[100]: <matplotlib.axes._subplots.AxesSubplot at 0x1e567d7c4c8>
```

Elizabeth Machumari stands out - No work between march - July!



9.0

Introduction to Machine Learning Models



INTRODUCTION TO MACHINE LEARNING MODELS

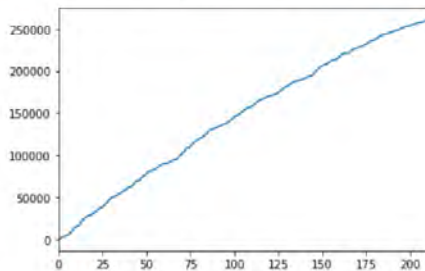
We will very quickly cover the most basic machine learning model – a linear regression model – which predicts how many kilograms of tea we expect to harvest every day based on the data provided.

Note this will not consider external factors such as weather, season, time of year, etc. That will come later!

First, let us rearrange our data into something that we can use. We only want a list of numbers to make our prediction on, so let's sum over all the different workers, make it a cumulative sum, and remove the date index since we don't really need that here.

```
In [328]: M model_data = df.sum(axis = 1).cumsum().reset_index(drop = True)
          model_data.plot(kind = 'line')
```

Out[328]: <matplotlib.axes._subplots.AxesSubplot at 0x21a51ac3608>



Great! So how do we put this data into a model?



Sklearn.

Remember when we said that there are libraries which make our lives easier? Well now is when we really start to see their benefits.

Sklearn has a specific function that is slightly different to Pandas, in that we must create an object before we can train it, rather than call everything at once like in Pandas.

Our approach will look like this:

- split the dataset up into two chunks: one to train the data on (that the model will see) and one to test the data (that the model will not see), and make all of our evaluation metrics with, including mean absolute error, root mean squared error, etc
- create a model object
- train the model
- test the model on the unseen data

```
In [326]: from sklearn import linear_model

In [329]: train = model_data[:-20]
          test = model_data[-20:]

In [330]: # create an 'untrained model'

          model = linear_model.LinearRegression(fit_intercept = True)

          # we can call the .fit() method, which will apply 'ordinary least squares' on our
          # points to optimise the predicted line

          model.fit(np.array(train.index).reshape(-1,1), train)

Out[330]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Done! Wasn't that simple?

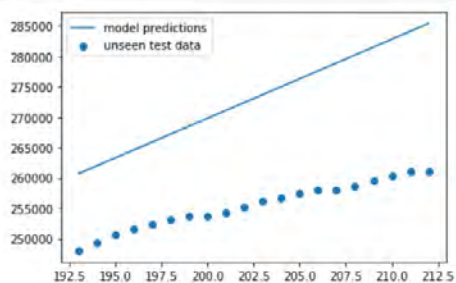


Now we can test the model and compare it to the real values in our test set:

```
In [331]: # test the model
predicted_values = model.predict(np.array(test.index).reshape(-1,1))
```

```
In [136]: # Let's plot these values to see how similar they are:
import matplotlib.pyplot as plt

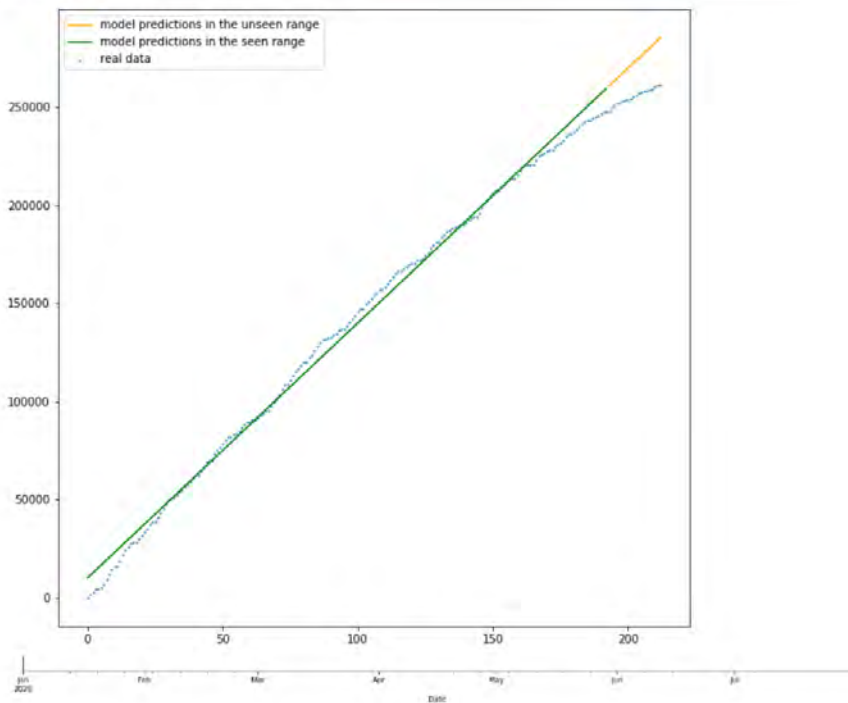
plt.figure()
plt.scatter(x = test.index, y = test, label = 'unseen test data')
plt.plot(test.index, predicted_values, label = 'model predictions')
plt.legend()
plt.show()
```



On first appearances, it doesn't look great! So, let's add this test data back into the train data to see why the results may look like this:

```
In [139]: in_sample_predictions = model.predict(np.array(train.index).reshape(-1,1))

plt.figure(figsize = (10,10))
plt.plot(list(test.index), list(predicted_values), label \
        = 'model predictions in the unseen range', color = 'orange')
plt.plot(list(train.index), list(in_sample_predictions), label = \
        'model predictions in the seen range', color = 'green')
plt.scatter(x = list(model_data.index), y = list(model_data), label = 'real data', s = 1)
plt.legend()
plt.show()
```



Now that we can see the full picture of what's going on in the model, we can understand why it may have performed so badly! Overall, the model fits the training data as expected, however since the model is only linear, it does not have the required complexity to model the drop-off in production which happens towards the end of the data.

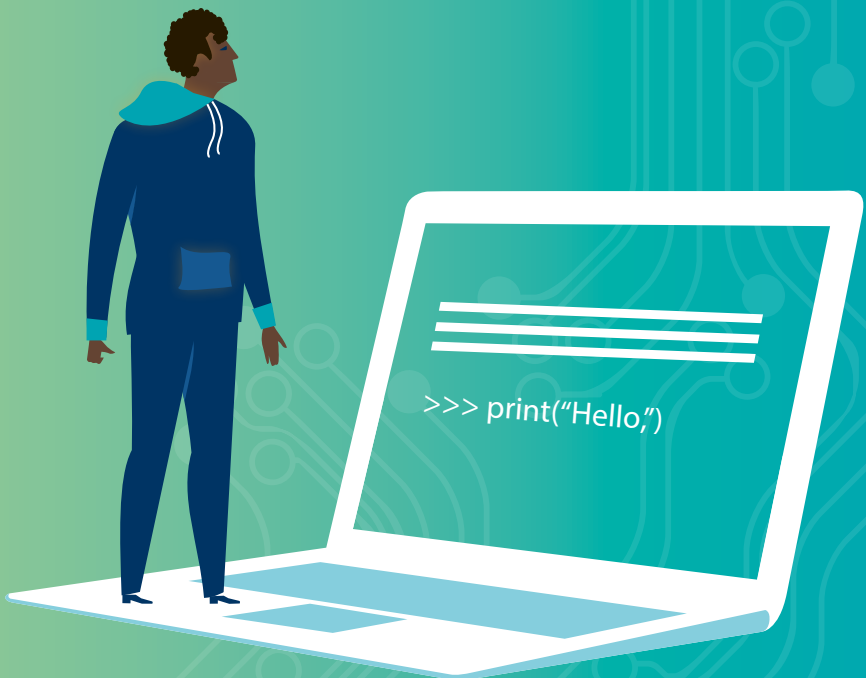
So how do we fix this?

Become a data scientist.



10.0

Can We Make a Model
That Accurately Predicts
the Amount of Tea
Plucked in Advance?



CAN WE MAKE A MODEL THAT ACCURATELY PREDICTS THE AMOUNT OF TEA PLUCKED IN ADVANCE?

Weather Data

We will be using weather data which has already been extracted via an API and saved as a .csv file. This makes our lives a little easier today, but in practice you will have to connect to an API to fetch this data. How to do this is covered in the interoperability workshops!

Weather data exploration - recap of previous lessons learnt

We should make a habit of exploring every new data set that we acquire in order to get a 'feel' for it, as this will inform us later in the modelling stage as to what is sensible to do and what isn't.

Why do we do data exploration if we can blindly throw data into a model?

The reason is that if you blindly throw data into a model, there will be no understanding of the outputs, and no understanding of how to improve the model! If the results are bad, that will be it: if they are good, that will be it. This is bad for you, or for anyone else that must trust the results your model is giving you – how can something be trusted if it is not understood?

Think of it like throwing darts whilst wearing a blindfold. Sure, you might hit the bullseye, but are you maximising your chances of doing so? Definitely not!

Description of remainder of the case study

We will be continuing with the Mapera Farm dataset; this time by focusing on the process in order to reach something useful. We'll also start with just two datasets, rather than describing the meaning of different bits of Python syntax.

Our three datasets that we will use in this course will be:

- the weather for the region the tea was plucked in using actual historical data
- the weather for the region the tea was plucked in using the daily average over the last 30 years
- the amount of tea plucked



First, we will explore the datasets we haven't seen before – Weather Data:

```
In [76]: # import our standard libraries
```

```
import pandas as pd
import numpy as np
```

```
In [111]: weather = pd.read_csv('Weather_data.csv')
```

```
In [112]: # a more convenient way to view data - by default pandas only displays a
# certain number of columns/ rows.

# we can overwrite that behaviour with the command below, useful if we know
# viewing the dataframe won't crash our pc

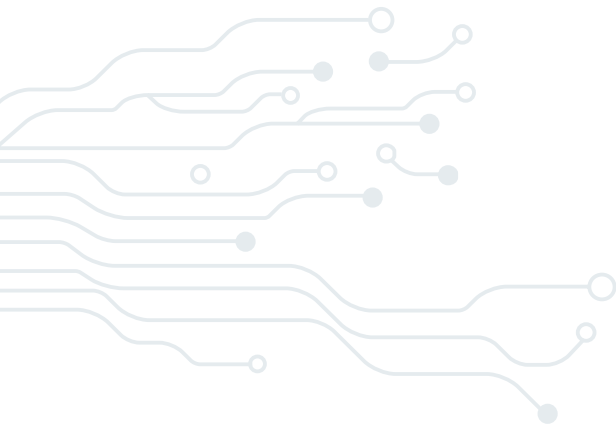
with pd.option_context('display.max_columns',None):
    display(weather.head())
```

	latitude	longitude	drivingDifficultyIndex	observationTimeUtcIso	precip1Hour	precip24Hour	precip6Hour	pressureChan
0	-1.13	36.71	NaN	2020-01-01T01:20:00+0000	0.0	0.0	0.0	-1
1	-1.13	36.71	NaN	2020-01-01T02:20:00+0000	0.0	0.0	0.0	1
2	-1.13	36.71	NaN	2020-01-01T03:20:00+0000	0.0	0.0	0.0	1
3	-1.13	36.71	NaN	2020-01-01T04:20:00+0000	0.0	0.0	0.0	1
4	-1.13	36.71	NaN	2020-01-01T05:20:00+0000	0.0	0.0	0.0	1

```
In [8]: weather.columns
```

```
Out[8]: Index(['latitude', 'longitude', 'drivingDifficultyIndex',
'observationTimeUtcIso', 'precip1Hour', 'precip24Hour', 'precip6Hour',
'pressureChange', 'relativeHumidity', 'wxPhraseCode', 'iconCode',
'snow1Hour', 'snow24Hour', 'snow6Hour', 'temperature',
'temperatureChange24Hour', 'temperatureDewPoint',
'temperatureFeelslike', 'temperatureMax24Hour', 'temperatureMin24Hour',
'uvIndex', 'visibility', 'windGust', 'windSpeed', 'Date', 'Datetime'],
dtype='object')
```

Note that the data is hourly and not daily like our tea-plucking data. We will need to resolve this!



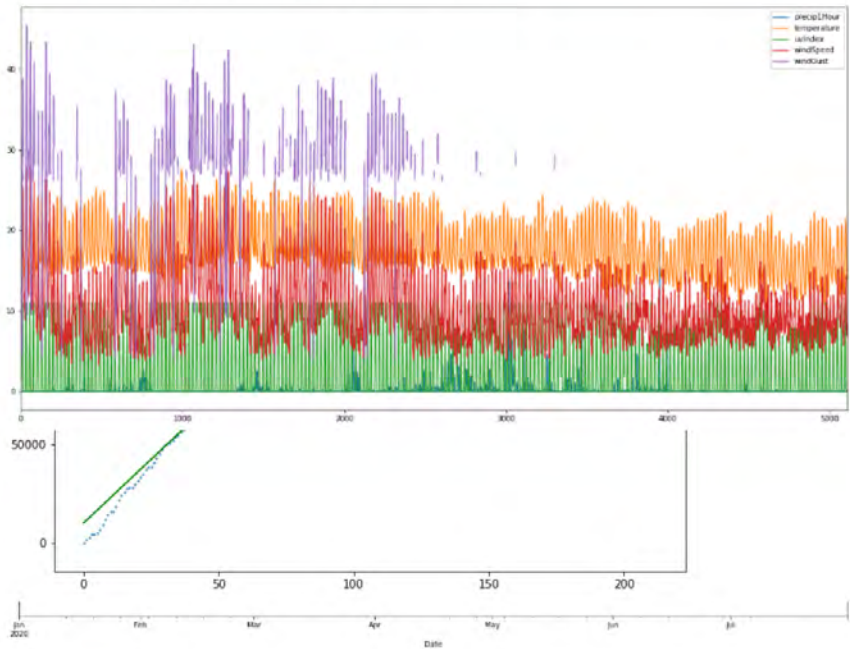

```
In [114]: # first thing's first, let's change the 'observationTime' to a real datetime instead of a
# timestamp

weather['Date'] = pd.to_datetime(weather['observationTimeUtcIso']).dt.date
weather['Datetime'] = pd.to_datetime(weather['observationTimeUtcIso'])

# Great, now let's see how precipitation, humidity, temperature, uvIndex and windspeed
# change with respect to each other

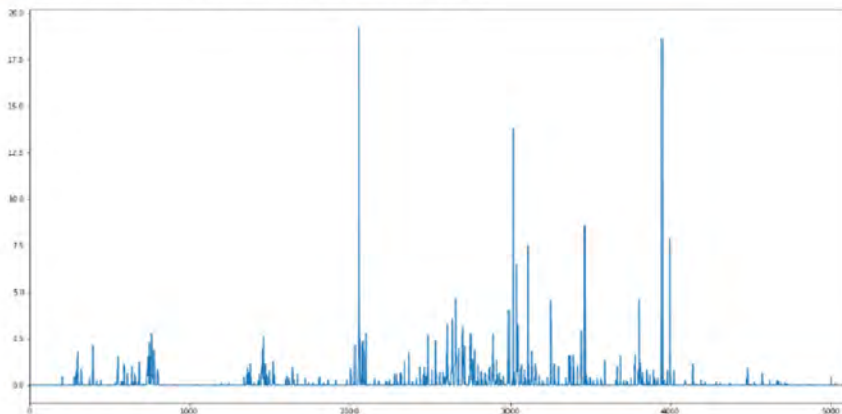
weather[['precip1Hour', 'temperature', 'uvIndex', 'windSpeed', 'windGust']].plot(figsize = (20,10))
```

Out[114]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc50c39188>



```
In [7]: weather['precipHour'].plot(figsize = (20,10))
```

```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc46f7c408>
```



Okay... so it may look pretty but studying this much data at once doesn't realistically help us to understand what is happening. At this point we have a few options:

- view fields one by one so we don't lose the ability to see data due to scaling/overlapping issues
- create an interactive plot so we can scroll and zoom across the data more easily
- transform the data into a format which is easier to work with

Since our tea-plucking numbers are measured by day, and this data is hourly, we should transform our weather data into daily values.

Let's define what we want to do for each of the features we've selected:

- `date`: we only want the first value, since they're all the same
- `gust`: sum (there are many null values, so if we took an average, a single gust would equal a day of gusting)
- `uvIndex`: sum (we want to know the total amount of sun that the tea gets)
- `temperature`: average (we want to know the daily average temperature)
- `precipHour`: sum (we want to know how much it rained in total that day)
- `windSpeed`: average (we want to know the average windspeed of the day)

To achieve this, we use the `'groupby()'` method (it has not been explicitly introduced in this course however is available in the Pandas' API documentation, followed by the `'agg()'` method).

```
In [138]: M day_weather = weather[['Date', 'windGust', 'uvIndex', 'temperature', 'precip1Hour', 'windSpeed']] \
          .groupby('Date'). \
          agg({'Date' : 'first', 'windGust' : 'sum', 'uvIndex' : 'sum', 'temperature' : 'mean',
              'precip1Hour' : 'sum', 'windSpeed' : 'mean'})
```

```
In [140]: M # we don't need the index (it was the date column)
day_weather.reset_index(drop = True, inplace = True)

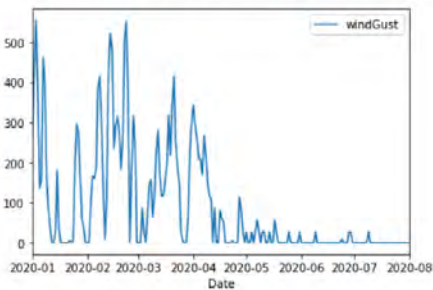
# show what it looks like.. Looks good!
day_weather.head()
```

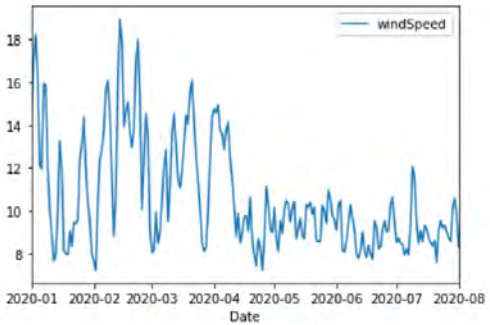
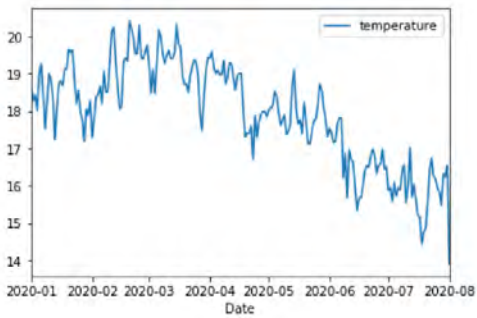
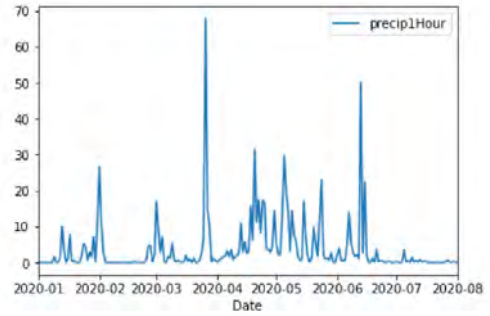
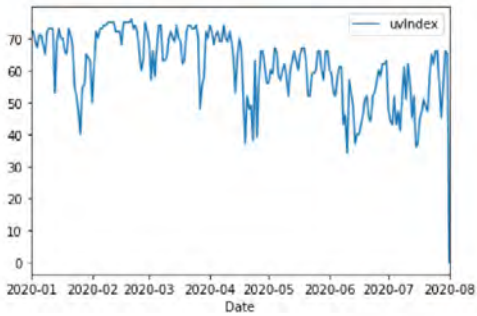
Out[140]:

	Date	windGust	uvindex	temperature	precip1Hour	wind Speed
0	2020-01-01	262.0	72.0	18.617391	0.0	14.339130
1	2020-01-02	432.3	72.0	18.275000	0.0	17.058333
2	2020-01-03	555.2	69.0	18.420833	0.0	18.212500
3	2020-01-04	373.1	67.0	18.008333	0.0	16.204167
4	2020-01-05	136.5	71.0	18.983333	0.0	12.095833

```
In [141]: M # let's just plot all the columns to see what they look like
```

```
for column in day_weather.columns[1:]:
    day_weather.plot(kind = 'line', x = 'Date', y = [column])
```





This looks like valuable information which would improve our model. The last values in `'uvIndex'` and `'temperature'` seem to look like data errors and may clip the dataset.



Weather Data API

Another source of data is an API which gives us the average daily temperature and precipitation on that day for the last 30 years combined. Let's tap into this and use it!

```
In [142]: # we can create our day and month codes and use this for the api
day_weather['month_code'] = pd.to_datetime(day_weather['Date']).dt.month.astype(str).str.zfill(2)
day_weather['day_code'] = pd.to_datetime(day_weather['Date']).dt.day.astype(str).str.zfill(2)

In [143]: day_weather

Out[143]:
```

	Date	windGust	uvindex	temperature	precip1Hour	windSpeed	month_code	day_code
0	2020-01-01	262.0	72.0	18.617391	0.00	14.339130	01	01
1	2020-01-02	432.3	72.0	18.275000	0.00	17.058333	01	02
2	2020-01-03	555.2	69.0	18.420833	0.00	18.212500	01	03
3	2020-01-04	373.1	67.0	18.008333	0.00	16.204167	01	04
4	2020-01-05	136.5	71.0	18.983333	0.00	12.095833	01	05
...
209	2020-07-28	0.0	45.0	15.470833	0.14	8.579167	07	28
210	2020-07-29	0.0	56.0	16.308333	0.00	10.166667	07	29
211	2020-07-30	0.0	66.0	16.245833	0.22	10.575000	07	30
212	2020-07-31	0.0	65.0	16.545833	0.12	9.833333	07	31
213	2020-08-01	0.0	0.0	13.900000	0.00	8.300000	08	01

214 rows x 8 columns

Now, we'll create some code which cycles through all our days and months and calls the weather API to return the previous values. This step may take some time, since calling an API is a time-consuming process, but once finished, we'll have the average temperature and rainfall values on a particular day for the last 30 years!

```
In [148]: import requests

temperature_holder = []
precipitation_holder = []

for i in range(len(day_weather)):
    day = day_weather.iloc[i, -1]
    month = day_weather.iloc[i, -2]

    call = ('https://api.weather.com/v3/wx/almanac/daily/1day?'
            'geocode=-1.133730,36.707264&format=json&units=h&'
            'day={}&month={}&apiKey=a6bba1f8c3a24f12bba1f8c3a28f12b1')

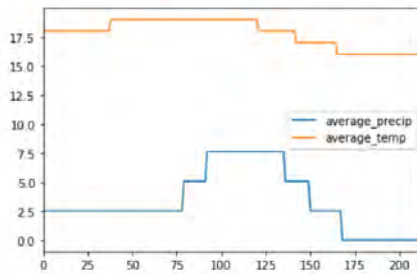
    response = requests.get(call.format(day, month))

    temperature_holder.append(response.json()['temperatureMean'][0])
    precipitation_holder.append(response.json()['precipitationAverage'][0])
```

```
In [149]: M day_weather['average_temp'] = pd.Series(temperature_holder)
day_weather['average_precip'] = pd.Series(precipitation_holder)

day_weather[['average_precip', 'average_temp']].plot()
```

Out[149]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc51756688>



```
In [150]: M # finally, let's look at what we've constructed so far:
```

```
day_weather
```

Out[150]:

	Date	windGust	uvIndex	temperature	precip1Hour	windSpeed	month_code	day_code	average_temp	average_precip
0	2020-01-01	262.0	72.0	18.617391	0.00	14.339130	01	01	18	2.54
1	2020-01-02	432.3	72.0	18.275000	0.00	17.058333	01	02	18	2.54
2	2020-01-03	555.2	69.0	18.420833	0.00	18.212500	01	03	18	2.54
3	2020-01-04	373.1	67.0	18.008333	0.00	16.204167	01	04	18	2.54
4	2020-01-05	136.5	71.0	18.983333	0.00	12.095833	01	05	18	2.54

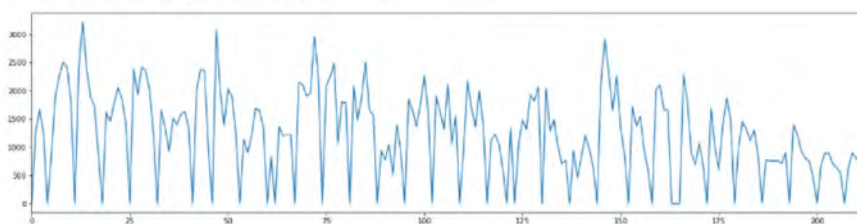
Reading in Mapera Farm Data

```
In [77]: df = pd.read_csv('mapera_farm_production_clean.csv')
```

Looking at Daily Mapera Farm Production

```
In [81]: df.iloc[:,1:].sum(axis = 1).plot(figsize = (20,5))
```

```
Out[81]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc4e55db48>
```



Notice how in using this version of the data, we can immediately see why our linear regression model struggled to accurately predict towards the end of the dataset: there was a clear downward trend in the amount of tea picked due to the seasons changing!

Also, is there a monthly trend? It looks like every fourth week, the amount plucked is lower than the previous three weeks. Let's plot weekly aggregates to see if this helps clear things up for us.

The first question that comes to mind is whether the number of workers (and specific workers) impacts the amount picked? If we can draw a correlation here, then it isn't so much about how fast the tea grows, but how effective the workers are at their job!

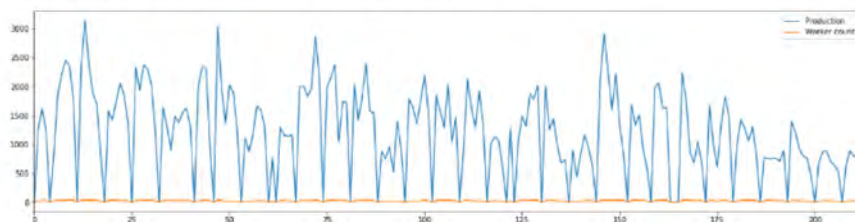



```
In [83]: M def counter(series):
    counter = 0
    for value in series:
        if value == 0:
            pass
        else:
            counter = counter + 1
    return counter

df['Production'] = df.iloc[:,1:-1].sum(axis = 1)
df['Worker count'] = df.iloc[:,1:-1].apply(lambda x: counter(x), axis = 1)
```

```
In [91]: M df[['Production', 'Worker count']].plot(figsize = (20,5))
```

```
Out[91]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc4e52ce48>
```



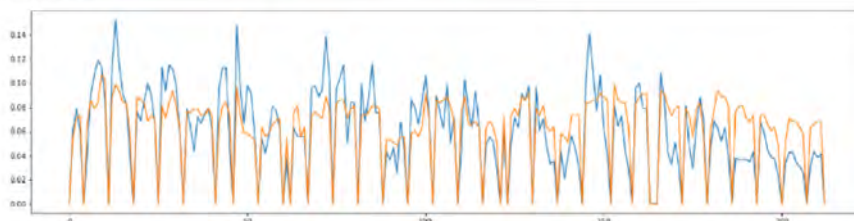
```
In [37]: M # we will just use the matplotlib library because these two columns have very different scales
# also let's import the preprocessing library to normalise our data, just so we can view it easier
```

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import preprocessing

Production = np.array(df['Production'])
normalised_p = preprocessing.normalize([Production])

Worker = np.array(df['Worker count'])
normalised_w = preprocessing.normalize([Worker])

plt.figure(figsize = (20,5))
plt.plot(range(len(normalised_p[0])), normalised_p[0])
plt.plot(range(len(normalised_w[0])), normalised_w[0])
plt.show()
```



We can see that there is a high correlation between the number of workers and the amount picked. But how far in advance do we know the number of workers we'll have?

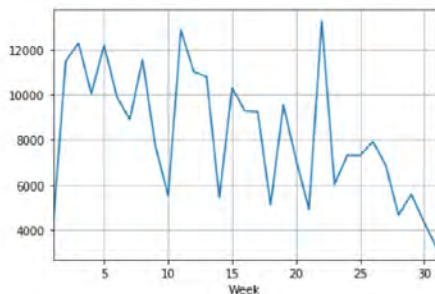
Also, the correlation breaks down in the most significant times: in the middle of the dataset when there is the most amount of tea to pick, and at the end of the dataset when even though there is a consistent workforce, not much tea has been picked.

Let's look at weekly production:

```
In [92]: M df['Week'] = pd.to_datetime(df['Date']).dt.week

In [94]: M to_model = df[['Date', 'Production', 'Week', 'Worker count']]
to_model['Date'] = pd.to_datetime(to_model['Date'])

In [95]: M to_model.groupby('Week')['Production'].sum().plot(grid = True)
Out[95]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc50015148>
```



There does appear to be a very clear pattern, whereby every month, there is a week in which production is low. We need to keep this in mind if we build a model, since our predictions will be more accurate with it than without!

Another thing is that the first and last weeks have a very low production rate. It is suspected that this is because of the way in which we have collected our data – the first and last weeks will be half-weeks which makes the values look unrealistically low.

If this is the case, then we should remove these weeks from our dataset since they do not fit the norm, and if our final model has a weekly component, then these weeks will suffer for it.

Also, what happened in week 22? Was this an anomaly? Or was it a legitimate amount of tea that was produced but not collected in the previous week? Understanding these questions can lead us to a more robust model.

Understanding why week 22 had such a large output:

Viewing the data below, we can immediately see that there was no ‘rest’ day in week 22. Tea was picked every day of the week.

This is either a mistake or there is a justifiable business-related reason for this, however:

- if there isn’t a good reason, it shouldn’t be in our model
- if there is a good reason, then we need to understand why this week had no break. Was it the end of the season? Was there an abnormally high amount of tea grown that week?
- if we understand the cause, then we can adapt our model for more accurate predictions.

```
In [44]: to_model.loc[to_model['Week'].isin([21, 22, 23])]
```

```
Out[44]:
```

	Date	Production	Week	Worker count	Rolling_week
138	2020-05-18	901.0	21	23	6005.0
139	2020-05-19	438.0	21	22	5184.0
140	2020-05-20	838.0	21	20	4574.0
141	2020-05-21	1164.0	21	29	4752.0
142	2020-05-22	948.0	21	29	5021.0
143	2020-05-23	627.0	21	29	4916.0
144	2020-05-24	0.0	21	0	4916.0
145	2020-05-25	2114.0	22	33	6129.0
146	2020-05-26	2915.0	22	33	8606.0
147	2020-05-27	2270.0	22	34	10038.0
148	2020-05-28	1599.0	22	34	10473.0
149	2020-05-29	2212.0	22	36	11737.0
150	2020-05-30	1311.0	22	35	12421.0
151	2020-05-31	833.0	22	34	13254.0
152	2020-06-01	0.0	23	0	11140.0
153	2020-06-02	1683.0	23	39	9908.0
154	2020-06-03	1335.0	23	34	8973.0
155	2020-06-04	1511.0	23	33	8885.0
156	2020-06-05	912.0	23	33	7585.0
157	2020-06-06	591.0	23	25	6865.0
158	2020-06-07	0.0	23	0	6032.0

Modelling

Developing a good model is an interactive approach. There should be a lot of trial and error in the process, backed up with logical, reasonable assumptions about both the data and the models used.

The steps taken in the modelling approach are generally as follows:

- develop a baseline model using what we have learnt in the data exploration stage
- see if we can improve on the baseline model

That's it!

So, to start, let's develop a baseline.

Baseline Model

A baseline model needs to be the most accurate thing that can be done in a short amount of time. Whatever that may be. The point of a baseline model is to have a reference point from which to benchmark your future models against, so that you know how well you are doing in comparison to something else. It is very difficult to understand how much better you have done if you don't have a reference point.

In some cases, a baseline model will just be the accuracy that is provided by competitors or other people. This is certainly the case in research. However, since there is nobody else modelling this problem, we will have to create our own.



Prophet

Three years ago, Facebook open-sourced a time-series prediction model of theirs called Prophet, and since then it has been widely accepted as a great out-of-the-box time series forecasting model. We may have more control over other models and there may be more complex modelling techniques which allow for higher overall accuracy, but as a starting point for a baseline model, this is exactly what we are looking for.

To start, we download it and install the library. In this case the library depends on another, called PyStan, so we download and install that one first (I hope you are getting used to this by now!).

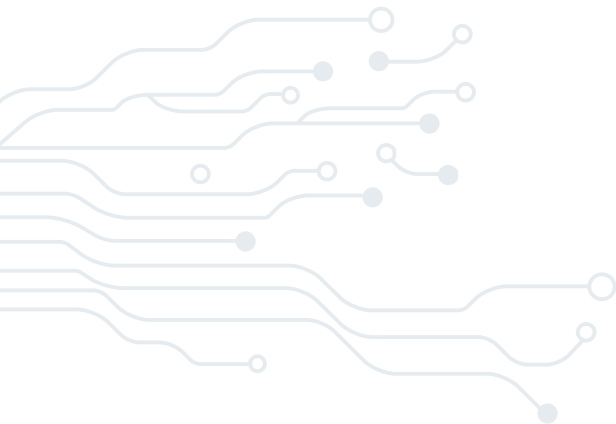
The easiest way to download Prophet is to use conda-forge, which is an alternative way of downloading libraries (basically a competitor to pip).

The command for this is `!conda install -c conda-forge fbprophet`. If you are having trouble with getting this to work, then try opening the Anaconda terminal in administrator mode.

```
In [99]: from fbprophet import Prophet
```

Now we just follow our steps to get to a model:

- create a train and test set
- create the model object and train it with the training data
- make predictions using the trained model
- merge the predictions back to our original dataset
- visualise the predictions and calculate accuracy metrics (mean absolute error)



```
In [100]: train = to_model.iloc[:-40,:2].rename(columns = {'Date' : 'ds', 'Production' : 'y'})
test = to_model.iloc[-40:,0:1].rename(columns = {'Date' : 'ds'})
full_set = to_model.iloc[:,0:1].rename(columns = {'Date' : 'ds'})
```

```
In [102]: to_model
```

```
Out[102]:
```

	Date	Production	Week	Worker count	Rolling_week
0	2020-01-01	0.0	1	0	NaN
1	2020-01-02	1272.0	1	22	NaN
2	2020-01-03	1626.0	1	28	NaN
3	2020-01-04	1248.0	1	29	NaN
4	2020-01-05	0.0	1	0	NaN
...
208	2020-07-27	641.0	31	25	4293.0
209	2020-07-28	895.0	31	26	4313.0
210	2020-07-29	796.0	31	27	4217.0
211	2020-07-30	857.0	31	27	4367.0
212	2020-07-31	0.0	31	0	3729.0

213 rows x 5 columns

```
In [101]: # create and train the model object
```

```
model = Prophet()
model.fit(train)
```

```
In [103]: # call the model on the test set
```

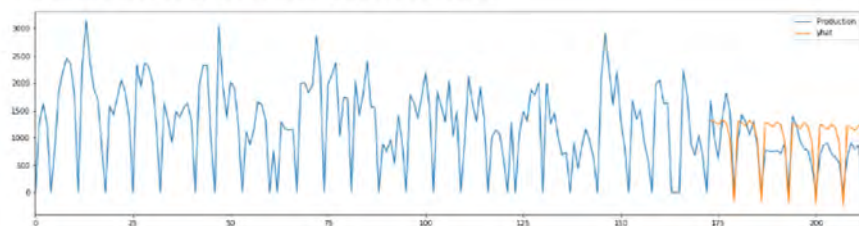
```
forecast = model.predict(test)

# select the relevant data we need
forecast = forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']]
```

```
In [63]: to_plot = to_model.merge(forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']],
                                how = 'left', left_on = 'Date', right_on = 'ds')
```

```
In [105]: to_plot[['Production', 'yhat']].plot(figsize = (20,5))
```

```
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc5068d248>
```



It doesn't look awful!

Let's use a metric to characterise our accuracy, or the mean absolute error. This way, we'll be able to know how many kilograms 'off' we are in our predictions, and to know immediately if our new model has improved on our baseline!

To calculate the mean absolute error, the equation is simple:

$$\frac{[\text{predictions} - \text{actuals}]}{\text{samples}}$$

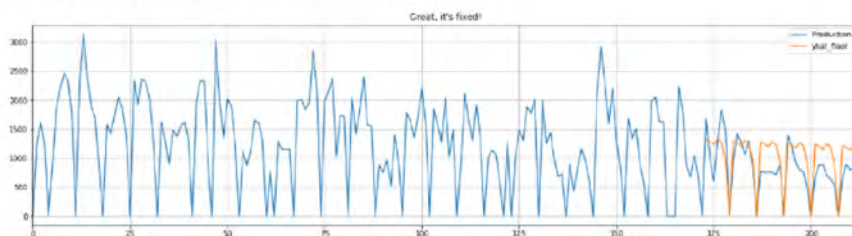
(P.s. our baseline predicts negative quantities for the days where there's meant to be zero – let's just fix this by adding a floor to our predictions).

```
In [106]: M def floor(value):
            if value < 0:
                return 0
            else:
                return value

            to_plot['yhat_floor'] = to_plot['yhat'].apply(lambda x: floor(x))
```

```
In [107]: M to_plot[['Production', 'yhat_floor']].plot(figsize = (20,5), title = "Great, it's fixed!", \
               grid = True)
```

```
Out[107]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc50b157c8>
```



```
In [110]: M length = len(to_plot.loc[~to_plot['yhat_floor'].isna()])

            mae = np.round(np.abs(to_plot['yhat_floor'] - to_plot['Production']).sum() / length, 2)

            print("Okay so our Mean absolute error appears to be {}. Let's see if we can improve on that".\
                  format(mae))
```

Okay so our Mean absolute error appears to be 323.75. Let's see if we can improve on that

Introducing Weather Data into the Model

With weather data, the idea is that we can explain the differences in tea plucked and therefore capture more of the nuances in the dataset, rather than relying entirely on previous prices. However, we need to be careful about how we use this data; if it isn't known more than one day in advance then we could be training a model using something that won't be available a month in advance.

We can also generate features from the individual workers on the farm. It may be that certain workers influence the total amount of tea picked in a given day.

We can add more variables to Prophet using the 'add regressor' option:

In [152]: `#Let's start by merging weather and productivity data`

```
day_weather['Date'] = pd.to_datetime(day_weather['Date'])
full_dataset = to_model.merge(day_weather, how = 'left', left_on = 'Date', right_on = 'Date')
full_dataset.rename(columns = {'Date': 'ds', 'Production' : 'y'}, inplace = True)
train_dataset = full_dataset.iloc[:40,:]
test_dataset = full_dataset.iloc[40:,:]
full_dataset.head()
```

Out[152]:

	ds	y	Week	Worker count	Rolling_week	windGust	uvIndex	temperature	precip1Hour	windSpeed	month_code	da
0	2020-01-01	0.0	1	0	NaN	262.0	72.0	18.617391	0.0	14.339130	01	
1	2020-01-02	1272.0	1	22	NaN	432.3	72.0	18.275000	0.0	17.058333	01	
2	2020-01-03	1626.0	1	28	NaN	555.2	69.0	18.420833	0.0	18.212500	01	
3	2020-01-04	1248.0	1	29	NaN	373.1	67.0	18.008333	0.0	16.204167	01	
4	2020-01-05	0.0	1	0	NaN	136.5	71.0	18.983333	0.0	12.095833	01	

In [153]: `model = Prophet()`

```
model.add_regressor('Worker count')
model.add_regressor('temperature', mode = 'multiplicative')
model.add_regressor('windGust', prior_scale = 3)
model.add_regressor('precip1Hour', mode = 'multiplicative')
model.add_regressor('average_temp', prior_scale = 10)
```

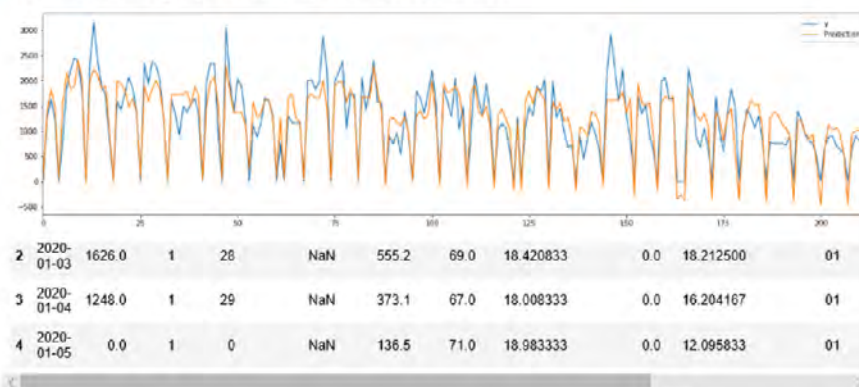
Out[153]: `<fbprophet.forecaster.Prophet at 0x1cc517b87c8>`

```
In [154]: M model.fit(train_dataset.loc[:,['ds','y', 'Worker count','temperature', 'windGust',
      'precip1Hour', 'average_temp']])
```

```
In [156]: M forecast = model.predict(full_dataset.loc[:,['ds','y', 'Worker count','temperature',
      'windGust', 'precip1Hour', 'average_temp']])
```

```
In [157]: M full_dataset['Prediction'] = list(forecast['yhat'])
full_dataset[['y', 'Prediction']].plot(figsize = (20,5))
```

```
Out[157]: <matplotlib.axes._subplots.AxesSubplot at 0x1cc5186df08>
```



Now this is starting to look like a real model! A lot of detail has been added here, and the best part is that the model hasn't been trained on the last four weeks of data, so this is what our test case looks like!

Let's add a floor to the data and see what our mean absolute error is now. In order to be fair, we take the floor of the test dataset, rather than the entire dataset, as the model has seen the entire training set before.

```
In [159]: test_forecast = model.predict(test_dataset.loc[:,['ds','y', 'Worker count',
                                                         'temperature', 'windGust',
                                                         'precip1Hour', 'average_temp']])

test_dataset['Prediction'] = list(test_forecast['yhat'])

# using our floor function we created earlier
test_dataset['Prediction_floor'] = test_dataset['Prediction'].apply(lambda x: floor(x))

In [160]: length = len(test_dataset.loc[~test_dataset['Prediction_floor'].isna()])

mae = np.round(np.abs(test_dataset['Prediction_floor'] - test_dataset['y']).sum()/ length,2)

print("Mean absolute error is now {} - a ~60% improvement on what we had before!".format(mae))

Mean absolute error is now 207.45 - a ~60% improvement on what we had before!
```

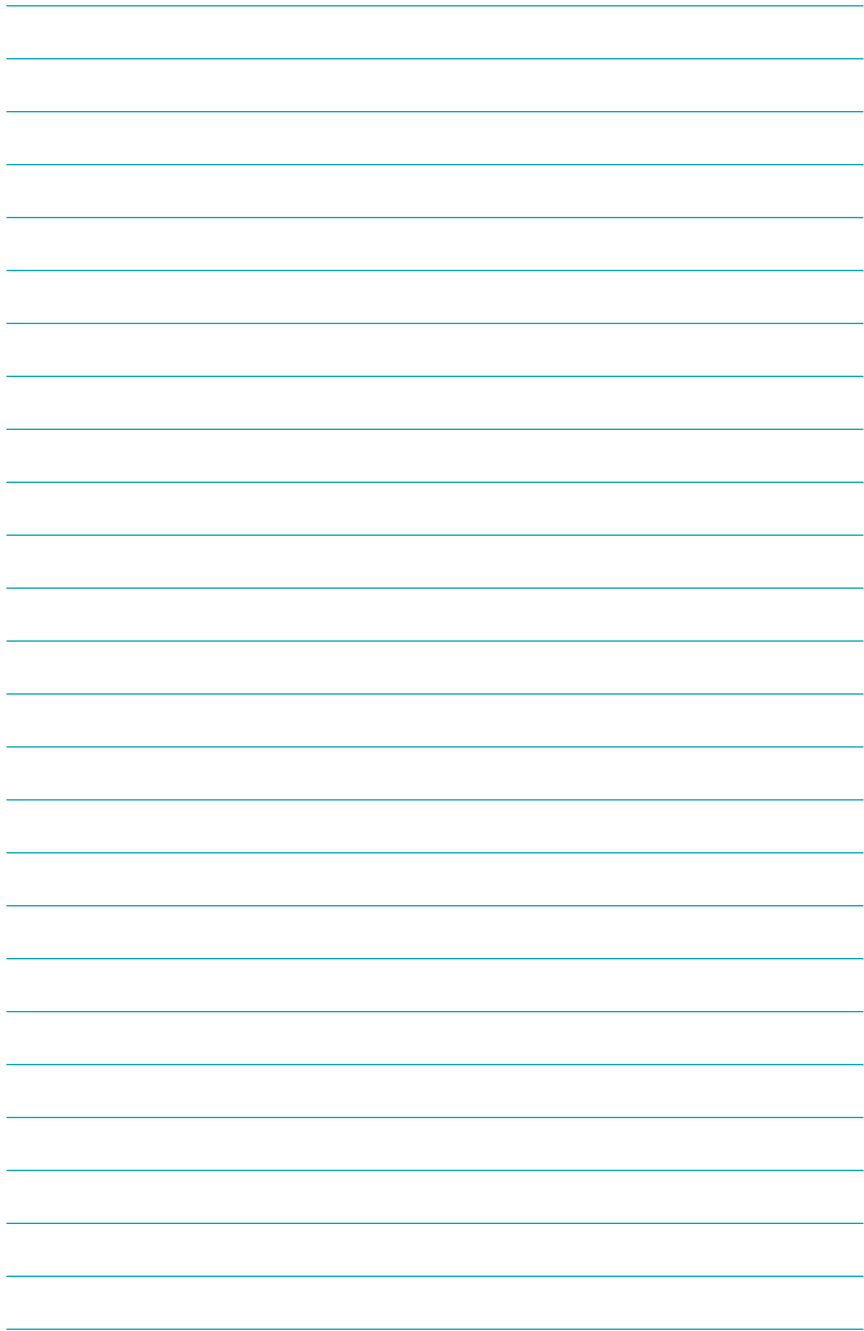
Next Steps

Next steps would be to understand where our model breaks down. There are significant cases where tea production is higher than expected, so it could be that a further combination of features derived from the weather data would illuminate this situation.



Notes

This image shows a single sheet of white paper with horizontal blue ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.



Publisher

Deutsche Gesellschaft für
Internationale Zusammenarbeit (GIZ) GmbH

Tech Entrepreneurship Initiative
“Make-IT in Africa”

Registered offices

Bonn and Eschborn

Friedrich-Ebert-Allee 32 + 36
53113 Bonn / Germany
T +49 228 44 60-0
F +49 228 44 60-17 66

Dag-Hammarskjöld-Weg 1-5
65760 Eschborn / Germany
T +49 61 96 79-0
F +49 61 96 79-11 15
E info@giz.de
I www.giz.de

Authors

Sergei Batishchev
Marc Hümmer
Silas Macharia
Desiree Winges

Editor

creative republic/David Steel

Design

creative republic, Frankfurt a. M. / Germany

Illustrations

© shutterstock & creative republic

GIZ is responsible for the content
of this publication

On behalf of the

German Federal Ministry for Economic
Cooperation and Development (BMZ)

As of

December 2020