Lab Session - Part 1

Jeroen de Haas

1 Introduction

This lab session we will acquaint ourselves with `pandas` — the *de facto* standard library for loading, inspecting and manipulating datasets. Knowing the terminology and concepts used in `pandas` is worthwhile. Many other data analysis tools and programming languages share, at least part, of the ideas introduced in this document.

Each lab session is offered as a Jupyter Notebook and a PDF. You are free to follow along using either. You are encouraged to execute every code cell, even those that are not part of an exercise. They configure the environment for later exercises. If you choose to follow the PDF, you will need to copy any code cells over.

2 Python, the language

Before we dive into pandas, let us first take a closer look at the Python programming language. We will introduce the concepts of

1. expressions,
2. variables,
3. classes,
4. and functions.

2.1 Expressions

Doing Data Science (or any programming) is doing computations. Prior to the arrival of artificial digital computers, the term "computer" referred to a human whose job it was to perform calculations. Nowadays the general term "computer" refers almost exclusively to that of an electronic device.

You have done a fair share of calculations in your life, from simple ones such as

$$1 + 1 \quad \text{or} \quad \frac{1}{2} \times \frac{1}{4}$$

to perhaps more complicated ones such as the definite integral

$$\int_{x=0}^{\infty} e^{-x} dx.$$

Now, you *will not* be confronted with those calculations in this course. "Don't panic!" But the integral does help us to get to grips with *how* computers work. Its calculation is not only more involved for humans, but for computers as well. Computers may be fast calculators, but their processors are restricted to the simplest of calculations: additions. More involved computations, like definite integrals, need to be broken down into a set of simple instructions that the processor can handle. That is what programming, at its core, is about.

At the core of computations are expressions, like $1 + 1$ or $\frac{1}{3} \times 21$, that can be reduced or evaluated to a value. $1 + 1 = 2$ and $\frac{1}{3} \times 21 = 7$. As our expressions grow more complex, or the numbers bigger, we tend to resort to a calculator.

Python can be used as a calculator. When provided with an expression in a form that it can understand, it will give its value as output. Try it for yourself! **Run the code in the block below and see the result**

```
1337 + 42
```

To add and subtract numbers you can use the `+` and `-` you are familiar with. Multiplication and division are done by the `*` and `/` operators respectively. The table below summarises the meaning of the most common operators in Python

Operator	Meaning	Example	Outcome
`+`	addition	`3 + 4`	`7`
`-`	subtraction	`8 - 3`	`5`
`*`	multiplication	`2 * 9`	`18`
`/`	division	`10 / 5`	`2`
`**`	exponentiation	`2 ** 3`	`8`

When faced with a more complex expression such as `2 + 1 * 5`, Python will use the rules of precedence that you are familiar with. The table above lists them in order of increasing precedence. That is, Python will first do any exponentiations, then divisions, then multiplications, *etc.* Parentheses can be used to change the order of evaluation.

Exercise: Study the expressions in the two code cells below. What do you expect their outcomes will be? Run the cell. Does Python's answer agree with your expectation?

```
5 + 1 * 6
```

```
6 * 6
```

We will now use Python to calculate a person's body mass index (BMI). The

formula for BMI is

$$\text{BMI} = \frac{\text{weight in kilograms}}{(\text{height in meters})^2}$$

Jeane is `1.76` meters tall and weighs `64` kilograms.

Exercise: Calculate Jeane's body mass index in one expression.

2.2 Variables

Single expressions are useful for a quick calculation. We enter an expression into our Python environment. Python will calculate the result, report it to us and then discard the result. Often, we want to build upon past results. For instance, we may wish to break a complicated expression into a number of smaller ones. To keep a value around, we need to assign it to a *variable*. We can then write the variable's name instead of the value.

The syntax of variable assignment is demonstrated in the following code block. There, a variable `myvar` is assigned the value `42`.

Exercise: Run the code cell.

```
myvar = 42
```

You may be surprised that Python *does not* tell us what value it assigned to a variable. We can however evaluate the variable by using it as an expression onto its own.

Exercise: Run the code cell below

```
myvar
```

Python will notice the variable `myvar` during evaluation and replace it with its value, `42` and then output its result to back to the environment.

2.3 Longer code cells

Thus far our code cells consisted of single expressions. We can group several expressions into a single cell, each on its own line. The following code cell performs three computations.

Exercise: Study the code cell. What do you expect its output will be? Now, run the code cell. Does its output agree with your expectation?

```
1 * 3
2 * 3
3 * 3
```

Notice how Python shows only the value of the last expression.

Exercise: Now write a code cell that does the following

1. Assign Jeane's height of 1.76 m to a variable `height`
2. Assign Jeane's weight of 7 kg to a variable named `weight`
3. Calculate her BMI using the two variables

2.3.1 Variable names

> There are only two hard things in Computer Science: cache invalidation and naming things. — Phil Karlton

The quotation above contains a joke. Cache invalidation is a tricky subject in computer science and is something best left to experienced software engineers. The second item, however, is relevant to novice and experienced programmers alike.

Python requires each variable name to start with a letter (e.g. a lowercase `a` or an uppercase `Z`) or an underscore (`_`). The rest of the variable name may also contain the digits `0` to `9`. Any other symbols may not appear in variable names. That means you cannot use spaces, exclamation points, question marks, brackets et cetera.

Exercise: Determine which of the following variable names are legal in Python. *Hint:* Try and see what happens when you assign a value to a variable with that name.

1. `A`
2. `a-number`
3. `num_people`
4. `num_people_who_watched_the_lord_of_the_rings_triology`
5. `alcohol%`
6. `#trending`
7. `visitors`

Not all names that conform to these rules are valid. Certain words are reserved by the Python language. These are known as *keywords*. Other words are used to refer to common instructions and should also be avoided. These names are coloured differently to stand out.

Exercise: Investigate the code cell below. Look at the color of each name and determine whether it is a legal variable name.

```
def
print
x
list
str
shopping_list
```

2.4 Classes

So far all our expressions involved numbers. Some of them, like Jeane's `height` contained a *fractional* or *decimal* part whereas others such as her weight (`74`) did not. These are two different *classes* of numbers. A number with a fractional part is known in Python as a `float`, whereas whole numbers are referred to as `int`s (from the mathematical term for whole numbers, *integer*). We can mix and match `float`s and `int`s in our programs. Python ensures the result belongs to the right class.

Note: The term *type* is used synonymously with class. We shall use the latter exclusively for the sake of consistency.

Another important class is that of `str` (short for *string*). Strings represent text and are written between single or double quotes. We have seen strings when we accessed columns by their name using the `loc`-attribute.

Exercise: Determine the class of each value in the code cell below.

```
3
3.14159
'Bonjour'
"Hello"
1.0
3 / 2
4 / 2
```

Exercise: Which of the following expressions are allowed? Try them in a code cell to check your answers

1. `3 + '3'` (*Note*: Mind the quotation marks)
2. `3 + 3`
3. `'three' + 'three'`
4. `3 * 'three'`
5. `'three' * 3`

2.5 Functions

The final concept is that of *functions*. Functions are like commands. When we call a function, we ask Python to perform a specific operation. We can think of a function call as a dog being told to "fetch the stick" by its owner.

The most famous function is `print`. It instructs Python to display a value to the user. It can be used throughout a code. Remember when we ran our first code cell with multiple lines? Python only reported the result of the last expression back to us. We can fix this using the `print` function.

Exercise: Run the next two code blocks and compare their output

```
1 * 3
2 * 3
3 * 3
```

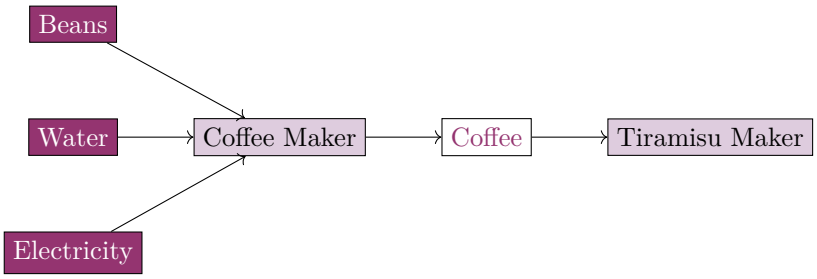
```
print(1 * 3)
print(2 * 3)
print(3 * 3)
```

In the last code cell we *call* the `print` function. The `print` function needs to be told *what to print*. This is done by placing a value or expression between parentheses immediately after the function name. That expression is known as an *argument* or *formal parameter*.

2.5.1 An analogy

Another way to conceptualise a function is as a device that performs a specific function. Consider a coffee maker. A coffee maker requires water, ground beans and electricity and uses these *inputs* to produce coffee. This is all we need to know to *use* the coffee maker. We need not know its internal workings.

Once we have our coffee, we can process it further. We could use this coffee in the preparation of tiramisu. So the output of one device, could be the input of another device. This process continues until we finally obtain our end result.



This is the pattern we will see again and again during our data analyses.

2.5.2 Multiple inputs

Like our coffee maker, functions may accept or even require more arguments. Here is an example that calls print with three arguments:

```
print('one', 'two', 'three')
```

Other functions that take a number of arguments are `max` and `min` which return the largest or smallest value among its arguments.

Exercise: Write a code cell that does the following:

1. Calculate the BMI for Camiel (1.83 m and 75 kg) and assign it to a variable.

2. Calculate the BMI for Thorsten (1.92 m and 80 kg) and assign it to another variable
3. Without looking at the results, find the value of the maximum and minimum BMI.

3 Manipulating Datasets With Pandas

We are now ready to start manipulating and analysing data sets using pandas. Pandas allows us to read data from a wide variety of sources including Excel sheets, CSV-files. As there is no standard way to store data sets in these file formats, we will occasionally need to provide pandas with additional information about the structure of the file.

First we will make pandas' functionality available to our environment. Conventionally, this is done with the following line of Python code.

```
import pandas as pd
```

There exist many other ways to import pandas, but this is the most common way. Once pandas has been imported, we can access its functionality under the `pd` prefix. To check if the import was successfully, we can access pandas' version number by running the next code cell:

```
pd.__version__
```

3.1 Meeting Our Data Set

This workshop we will be working with a dataset by the Central Bureau for Statistics on chicken farms in the province of Noord-Brabant (Centraal Bureau voor de Statistiek 2017). A copy of this dataset should have been included with this document under the name `chickenfarms.xlsx`.

Exercise: Open `chickenfarms.xlsx` in a spreadsheet editor. Investigate its structure.

3.2 Loading the Data Set

Pandas offers a variety of functions for reading and writing Excel files. For now, we will concentrate on the function `read_excel`. We can call this function and pass the file name as a string. In order to access the function, we prefix its name with `pd`.

Exercise: Run the code cell below. Inspect its output, do you feel the output is usable? How should the output, ideally, look like.

```
pd.read_excel('chickenfarms.xls')
```

Exercise: Study the file once more in your spreadsheet editor. Determine which row contains the column names and which column contains the row names.

3.2.1 Counting Like a Computer

After inspection we find that the *second row* contains the column labels. The row labels are found in the *first column*. The values are stored in the second and third column from the third row on down. The data ends after row 64.

As humans, we are used to start counting from 1. Computers however tend to work with *indices* that start at 0! What we would see as the first column, a computer considers the column at index 0. **Remember:** Humans start counting at 1, computers at 0.

Exercise: Now study the documentation of `read_excel` by clicking its name. You will find a list of the function's formal parameters. *Don't panic!* If the explanation of these variables is not clear, that is perfectly fine. Find the parameters that control where pandas looks for the row and column labels. What values would you pass to these parameters?

Exercise: The `nrows` parameter controls how many rows it will read from the header row on down. Determine a suitable value for this parameters.

3.2.2 Specifying Values for Parameters

We now have determined suitable values for the following parameters:

- `header`
- `index_col`
- `nrows`

At the top of the documentation of `read_excel`, we find a short summary of all parameters and their default values.

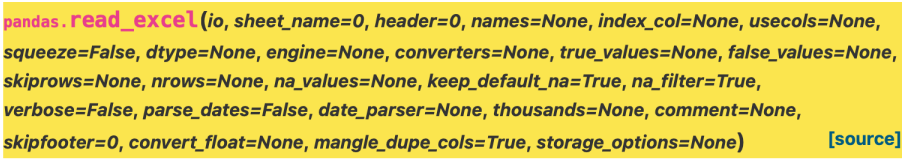


Figure 1: The parameters for `read_excel` and their default values

Notice that the first parameter has no default value. This means we *must always* specify its value when calling the function. This makes sense. How could pandas load data from a file without knowing its file name. The file name will always be the first value we pass to `read_excel`. Next, we specify the value for the other, *optional*, arguments. We do this, by placing the parameter name followed by an equals sign and the desired value between parentheses, like so:

```
pd.read_excel('chickenfarms.xls', nrows=10, header=0, index_col=0)
```

Exercise: In the code cell below write the appropriate function call by correcting the parameter values from the example.

```
# Your answer here
```

If successful, `read_excel` will produce a value, also known as an *object* of the class `DataFrame`. You can think of `DataFrame` much like other classes such as `int` and `str`. The `DataFrame` class is designed to represent and manipulate large data sets.

3.2.3 Keeping Our `DataFrame` Around

When Python executes the call to `read_excel`, it produces the resulting value, displays the result to us and then discards the result. To keep it around, we need to assign the result to a variable. A typical name for a variable that refers to a `DataFrame` is `df`.

Exercise: In the cell below, write code that calls `read_excel` and assigns the result to the variable `df`.

```
# Your answer here
```

3.3 Obtaining Information About a DataFrame

Now that we have a reference to our `DataFrame` in the form of our variable `df`, we can ask its attributes and call functions that are designed to work on this particular `DataFrame`. Such functions are known as `methods`. We can call them, much like functions from a module, by prefixing the method name with the variable name followed by a dot, *i.e.,* in our case, `df.`.

Exercise: To call the describe method on our `DataFrame`, run the following code cell.

```
df.describe()
```

There exist many more methods that we can call.

Exercise: Study the documentation of `DataFrame.info`. Which parameters of this method are required? Which are optional? In the next code cell, call the `info` method leaving all optional parameters at their default value.

```
# Your answer here
```

Exercise: The `info` method has an optional parameter named `verbose`. It expects a value of the `bool` class which we have not studied yet. There exist two valid values for this class: `True` and `False`. Try calling `info` with `verbose=True` and `verbose=False`. How does the output differ?

```
# Your answer here
```

```
# Your answer here
```

Other useful methods include - `DataFrame.head` - `DataFrame.tail`

Exercise: Study the documentation of these methods and write a code cell that returns the top five rows and another that returns the last ten rows.

```
# Your answer here
```

```
# Your answer here
```

3.4 Accessing Attributes

We have seen how we can call methods on an object by using the dot-notation. We use the same notation to access an object's *attributes*. We can think of attributes as *properties* or *characteristics* of an object.

When describing a dog in terms of its characteristics, we may state its breed, age, name and sex. The attributes of a `DataFrame`s include its `shape` (the number of rows and columns) and the labels of its rows and columns.

The code block below illustrates how to access the number of rows and columns in a `DataFrame` using the `shape` attribute:

```
df.shape
```

3.5 Changing Column Labels

Presently, the column labels of our `DataFrame` are in Dutch and, admittedly, not convenient to type. We will therefore shorten them to appropriate English labels.

To do this, we need to change one of the *attributes* of our `DataFrame`. This attribute is `columns` and it contains a *list of column labels*. We can change the labels by assigning a new list to the attribute. Attribute assignment shares its syntax with variable assignment and should feel familiar:

```
df.columns = ['farms', 'chickens']
```

Exercise: Run the code cell above and then call the `describe` method on the `DataFrame`, do you notice a difference?

```
# Your answer here
```

This block changes the column labels to `farms` and `chickens` reflecting that the first contains the number of chicken farms within a municipality and the other contains the total number of chickens held by those farms.

The expression on the right hand side of the equals sign evaluates to a list of two strings. Lists are written as a comma separated list of values surrounded by square braces. Any number of values (including zero) is acceptable.

Exercise: In the code cells below create

1. an empty list,
2. a list containing just your name as a string
3. a list containing the sequence, `3`, `1`, `4`, `1` and `5`.

```
# Your answer here
```

```
# Your answer here
```

```
# Your answer here
```

3.6 Accessing Values

One of the most basic operations we can do on `DataFrame`s is accessing observations and values. Pandas offers at least three different ways to access values, each with their own peculiarities. We will therefore restrict ourselves to the `loc`-attribute.

The `loc`-attribute can be used to select values using the row and column labels.

1. To select a single value pass the `loc`-attribute a row label and column label separated by a comma:

   ```
   df.loc['Altena', 'farms']
   ```

2. To a select a single row, use a colon (`:`) as the column label. The colon signifies all elements across a given dimension, e.g.,

   ```
   df.loc['Altena',:]
   ```

3. To select a column, use a colon (`:`) for the row name

   ```
   df.loc[:, 'farms']
   ```

EXERCISE: Write a code cell that calculates the average number of chickens per farm in Bergeijk. Use the `loc`-attribute twice. Once to obtain the total number of chickens, and another time to select the number of farms.

```
# Your answer here
```

3.6.1 Using Multiple Labels

Both the row and column selector of the `loc`-attribute accept lists as their arguments. This allows us to extract multiple rows in a single expression.

Recall that a list is written as a comma separated list of values surrounded by brackets, *i.e.,* `[` and `]`. The next code cell illustrates how to select the number of farms in three municipalities:

```
df.loc[['Helmond', 'Hilvarenbeek', 'Son en Breugel'], 'farms']
```

Exercise: Which of the following municipalities has the most chickens? Answer your question by selecting the required data from the `DataFrame` with a single use of the `loc`-attribute. You may inspect yourself to determine the answer

1. Geertruidenberg
2. Steenbergen
3. Moerdijk

```
# Your answer here
```

3.7 Series

The `DataFrame` class is used for objects that represent a data set with rows and columns. It is a two dimensional structure. If we now extract a row or column from the `DataFrame`, we are left with a single or column, a one dimensional structure. These structures belong to `Series` class.

Let us extract the two columns using the `loc` attribute and assign them to variables `farms` and `chickens`:

```
farms = df.loc[:, 'farms']
chickens = df.loc[:, 'chickens']
```

Exercise: Execute the following code cell and use the next two cells to call the describe method on these variables.

```
# Your answer here
```

```
# Your answer here
```

Observe that although `Series` and `DataFrame`s are different classes, they have a number of attributes and methods in common.

3.7.1 Calculating Statistics

We can use our `Series` objects to answer questions such as:

1. What is the largest number of farms in a single municipality?
2. How many chickens were commercially held in the province of Noord-Brabant in 2017?
3. Which municipality has the fewest chickens?
4. What are the five municipalities with the largest number of chickens.

Exercise: Study the following methods of the `Series` class. Use them to answer the three questions listed above.

- `Series.min`
- `Series.max`
- `Series.sum`
- `Series.idxmin`
- `Series.idxmax`
- `Series.nlargest`
- `Series.nsmallest`
- `Series.index`

```
# Your answer here
```

```
# Your answer here
```

```
# Your answer here
```

3.8 Visualising Series

`Series` have methods that allow us to visually expect their values. We will concentrate on the `Series.plot` method which allows us to control various aspects of the visualisation.

We can control the type of visualisation via the `kind` parameter. To create a bar plot of the farms in Noord-Brabant we could write:

```
farms.plot(kind='bar')
```

Unfortunately, the names of the municipalities are hard to read. Their names are too close together. Let us fix this by changing the size of the figure via the `figsize` parameter. This parameter requires two numbers, one determines the width of our figure, the other the height. How can we achieve that? A possibility is to provide a *list of two `int`s or `float`s!

```
farms.plot(kind='bar', figsize=[12,6])
```

If the text is too small, we can change its size using the `fontsize` parameter. To accommodate for this larger font size, we will increase the width of the figure even further.

```
farms.plot(kind='bar', figsize=[24,6],fontsize=18)
```

If the number of parameters grows large, the readability of our code may be affected. You can spread the parameters over multiple lines by inserting a new line (e.g., by pressing return) after a comma. The behaviour of the following code cell is identical:

```
farms.plot(kind='bar',
           figsize=[24,6],
           fontsize=18)
```

Exercise: Study the documentation of `Series.plot`. Find a call to the `plot` method that replicates the next figure:

```
# Your answer here
```

Exercise: Create a histogram of the number of farms in Noord-Brabant. The figure should resemble

```
# Your answer here
```

3.9 Using Series in Calculations

Suppose now that we wish to calculate the average number of chickens per farm for every municipality. This means that we should divide the number of chickens

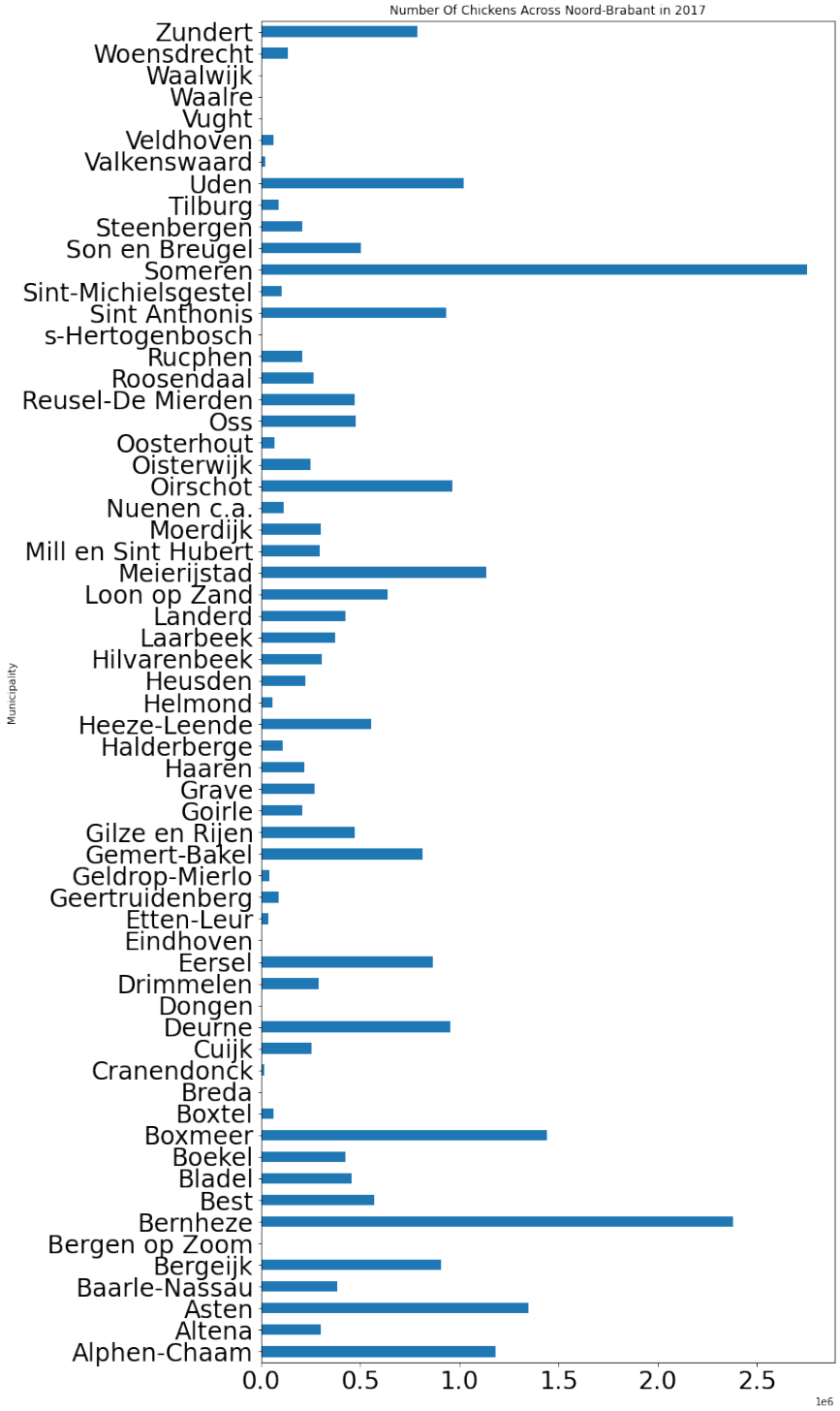


Figure 2: A horizontal bar plot of the number of chickens across the municipalities in Noord-Brabant

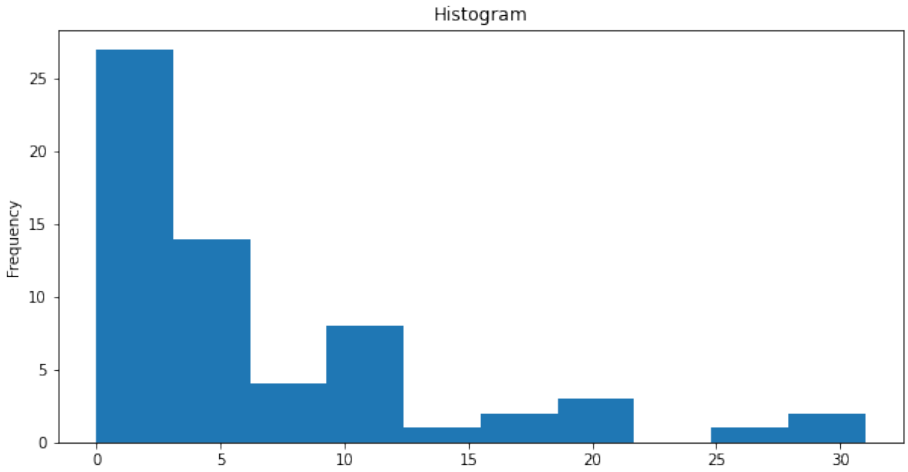


Figure 3: A histogram of the number of farms in the municipality

in a municipality by the number of farms. For this we can use the same operators we used in our first expressions.

As an example, we wil consider the expression `farms + chickens`. This expression asks Python to create a new Series for every municipality where the values contain the sum of the farms and chickens in that municipality.

Exercise: Create a `Series` that represents the average number of chickens per farm in each municipality. Assign your result to the variable `avgchick`. Next, call the describe method on that variable.

```
# Your answer here
```

We can add columns or rows to our `DataFrame`. To this end, we use the `loc` attribute to select a non-existent column and then assign a `Series` object to it:

```
df.loc[:, 'avgchick'] = avgchick
df.columns
```

As you can see, the `columns` attribute now contains an additional label for our newly added column.

Exercise: A quantity of central importance in the poultry industry is the Municipal Squared Chicken Score (MSCS) calculated by the following formula. *Add a column named* `mscs` *that contains this quantity*

$$\text{MSCS} = \frac{\text{number of farms in municipality}}{(\text{millions of chickens in municipality})^2}$$

```
# Your answer here
```

HINT: You can multiply or divide all values in a Series by the same value (or add or subtract a number) by applying the standard operators to a `Series` and a number. This creates a new `Series` and thus *does not affect* the values in the original `Series`.

```
# Add a 1000 to every value in the series
farms + 1000
```

Exercise: Which municipalities have the largest and smallest MSCS? What are their scores?

```
# Your answer here
```

```
# Your answer here
```

Exercise: Are there any unusual values in the new column? Can you explain them?

3.10 Selection Using Masks

Thus far, we have used the `loc`-attribute to extract entire rows or columns based on their labels. Certain analyses may require a more fine grained selection. We may for instance be interested in a subset of our `DataFrame` such as all the municipalities with fewer than five chicken farms. This is done in a two step procedure. First we create a *mask* and pass that mask to the `loc`-attribute.

A mask is a `Series` of `bool` values. Objects of this class are either `True` or `False`. If we want to select a subset from a `Series` of 10 values, we pass a mask of 10 `bool`s to its `loc`-attribute. The result is a `Series` of those values for which the mask was `True`. Values for which it is `False` are discarded.

Let us study an example. The following code cell creates a mask that is `True` if a municipality has no chicken farms:

```
farms == 0
```

Exercise: Run the cell and observe its results.

Exercise: Python uses double equal signs to check for equality, a convention shared by many programming languages. Can you explain why this is? *Hint:* Imagine that the code cell above contained a single equal sign. How would that affect the meaning of the code?

Since we did not assign the result to a variable, Python has discarded the mask. In the next code cell, we create the mask and then use it to select all information for these municipalities.

Exercise: Run the following cell.

```
no_farms = farms == 0
df.loc[no_farms, :]
```

The expression `df.loc[no_farms, :]` should be read as: "from `DataFrame` `df`, for all rows for which `no_farms` is true, select all columns (signified by the colon `:`). It is also possible, but not common, to select columns based on a mask.

The most used comparison operators are summarised in the table below

Operator	Meaning	Example
`==`	equals	`df.loc[:, 'farms'] == 1`
`!=`	does not equal	`df.loc[:, 'farms'] != 0`
`>`	greater than	`df.loc[:, 'farms'] > 17`
`>=`	greater than or equal	`df.loc[:, 'farms'] >= 18`
`<`	less than	`df.loc[:, 'farms'] < 17`
`<=`	less than or equal	`df.loc[:, 'farms'] <= 16`

Exercise: How many chickens are held in total by all municipalities with less than 10 farms?

```
# Your answer here
```

3.10.1 Combining Masks

We can combine several masks to create more complex selection criteria. We may for instance create a mask to find the municipalities with more than 5 but less than 10 farms:

```
(farms > 5) & (farms < 10)
```

The ampersand, `&`, stands for *and*, a municipality with more than 5 but less than 10 farms must meet both the first *and* second criterion. Note that the mask expressions are placed within parentheses. Failure to do will lead to errors.

The three operators that operate on masks are shown in the table below.

Operator	Meaning	Example
`&`	Matches those selected by both masks	`a & b`
`|`	Matches anything that matches at least one of the masks	`a | b`
`~`	Matches anything the original mask did not select	`~a`

Exercise: Find all municipalities that have 0, 1, or more than 25 farms.

```
# Your answer here
```

4 Homework

4.1 Oh no, more chickens!

Exercise: Of all municipalities with 10 or more chicken farms, which municipality has the highest number of chickens per farm?

```
# Your answer here
```

Exercise: Determine the average number of chickens of the municipalities with more than $700,000$ but less than $5,000,000$ chickens. In which of those municipalities is the number of chickens below this average?

```
# Your answer here
```

Exercise: The method `DataFrame.plot` can be used to visualise more than one column in a DataFrame. Study its documentation and create a scatter plot to illustrate the relation between the number of farms and the number of chickens. Try to get as close as possible to the figure below, or make it even better!

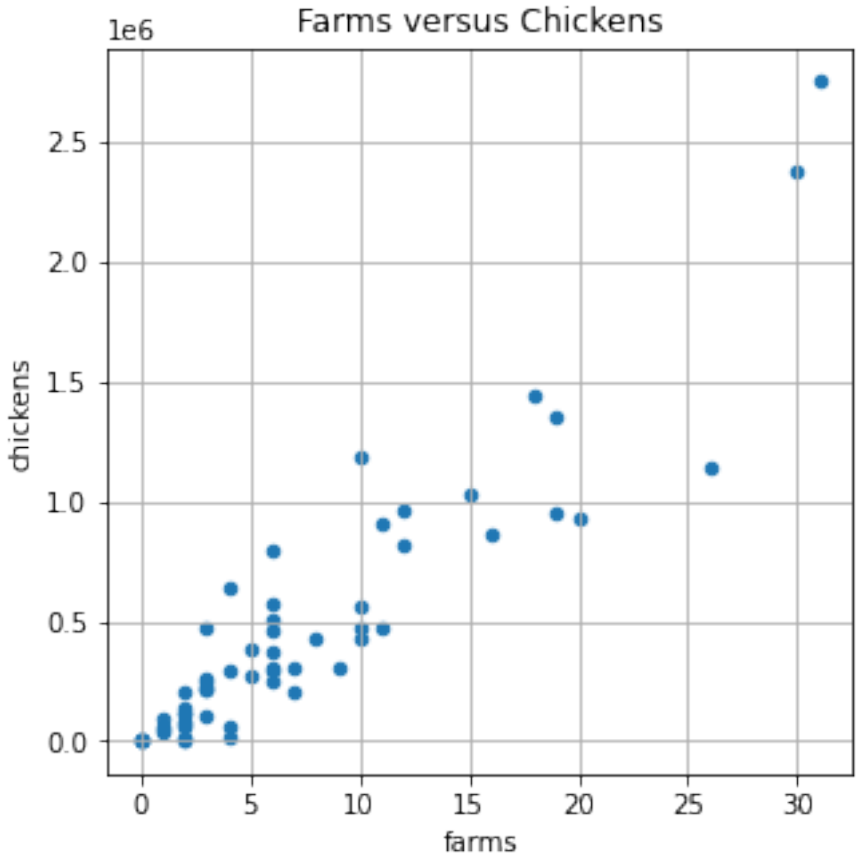


Figure 4: Example scatter plot

```
# Your answer here
```

Exercise: In this exercise you will add a new column to the `DataFrame df`. Give it the name `category`. The value of this column depends on the number of farms in the municipality. The table below shows this relation between category and number of farms.

Category	Definition	Meaning
0	farms $= 0$	No farms
1	$0 <$ farms < 2	1 farm
2	$2 \leq$ farms < 5	2, 3 or 4 farms
3	$5 \leq$ farms < 11	5 up to 10 inclusive farms
4	farms ≥ 11	more than 10 farms

You can verify your solution comparing your results with those in the table below

Municipality	Category
Waalre	0
Geldrop-Mierlo	1
Veldhoven	2
Cuijk	3
Uden	4

Hint: The code cell below illustrates how to add a new column of all tens. Change this value to something more appropriate and use it as a starting point.

```
df.loc[:, 'category'] = 10
```

```
# Your answer here
```

```
# Your answer here
```

4.2 Something More Powerful

In this exercise we are going to look at a new dataset of the number of solar power installations across the Netherlands (Centraal Bureau voor de Statistiek 2021). The original data set has been translated to English for use in this course.

It contains data at four levels. The level is indicated by its Dutch label in the column `RegionType`.

1. Nationwide (Dutch: *Land*)
2. Municipality (Dutch: *Gemeente*)
3. District (Dutch: *Wijk*)
4. Neighbourhood (Dutch: *Buurt*)

Each entity is assigned a unique code. Each municipality is assigned a four digit number prefixed by `GM`, e.g., 's-Hertogenbosch is assigned code `GM0796`. District codes are prefixed by `WK` followed by the four digits of the municipality code and two digits unique to the district. Neighbourhoods add another two digits and are prefixed by `BU`.

The column `Installations` reports the total number of solar installations at the specified level. The column `Power` indicates the combined power in kilowatts (kW).

Exercise: The data is offered as a CSV-file named `solar.csv`. Unfortunately, values are not separated by semicolons instead of commas. By default, pandas expects the latter. Do the following:

1. Search the documentation of [`pandas.read_csv`] for the parameter that controls the expected separator.
2. Figure out how to use the first column (named `Code`) as the index, i.e., the row labels.
3. Load the data from `solar.csv` and assign it to the variable `solar_df`

```
# Your answer here
```

Exercise: Which municipality has the most solar installations? How many installations are present there.

```
# Your answer here
```

Exercise: What do you think of the following claim: "The answer to the previous exercise reveals which municipality has the most environmentally conscious population"

Exercise: Find code of the neighbourhood which produces the most power per installation.

```
# Your answer here
```

Exercise: Use the method `DataFrame.nsmallest` to find the ten municipalities with the least solar installations.

```
# Your answer here
```

Exercise: Create a scatter plot that shows the relation between the number of installations and total power at the level of districts

```
# Your answer here
```

Exercise: Which neighbourhoods have more than 4000 installations?

```
# Your answer here
```

Bibliography

Centraal Bureau voor de Statistiek. 2017. “Veeteelt - Gemeenten Noord-Brabant.” https://brabant.databank.nl/.

———. 2021. “Zonnestroom; Wijken En Buurten, 2019.” https://opendata.cbs.nl/statline/portal.html?_la=nl&_catalog=CBS&tableId=85010NED&_theme=282.