Data Frames 1

June 28, 2022

Question

- How is data read into a dataframe?
- What are different ways to manipulate data in dataframes?
- What makes data visualisation simple in Python?

Objectives

- Import data set as Pandas dataframe
- Inspect data frame and access data
- Produce an overview of data features
- Create data plots using Matplotlib

1 Prerequisites

- Indexing of Arrays
- For Loop through Array
- Basic Statistics (distributions, mean, median, standard deviation)

The diabetes data set is the challenging task.

2 Challenge: The diabetes data set

Here is a screenshot of the so-called diabetes data set. It is taken from this webpage and it is one of the example data sets used to illustrate machine learning functionality in scikit-learn (Part III and Part IV of the course).

www4.stat.ncsu.edu/~boos/var.select/diabetes.tab.txt

AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6	Y
59	2	32.1	101	157	93.2	38	4	4.8598	87	151
48	1	21.6	87	183	103.2	70	3	3.8918	69	75
72	2	30.5	93	156	93.6	41	4	4.6728	85	141
24	1	25.3	84	198	131.4	40	5	4.8903	89	206
50	1	23	101	192	125.4	52	4	4.2905	80	135
23	1	22.6	89	139	64.8	61	2	4.1897	68	97
36	2	22	90	160	99.6	50	3	3.9512	82	138
66	2	26.2	114	255	185	56	4.55	4.2485	92	63
60	2	32.1	83	179	119.4	42	4	4.4773	94	110
29	1	30	85	180	93.4	43	4	5.3845	88	310
22	1	18.6	97	114	57.6	46	2	3.9512	83	101
56	2	28	85	184	144.8	32	6	3.5835	77	69
53	1	23.7	92	186	109.2	62	3	4.3041	81	179
50	2	26.2	97	186	105.4	49	4	5.0626	88	185
61	1	24	91	202	115.4	72	3	4.2905	73	118
34	2	24.7	118	254	184.2	39	7	5.037	81	171
47	1	30.3	109	207	100.2	70	3	5.2149	98	166
68	2	27.5	111	214	147	39	5	4.9416	91	144
38	1	25.4	84	162	103	42	4	4.4427	87	97
41	1	24.7	83	187	108.2	60	3	4.5433	78	168
35	1	21.1	82	156	87.8	50	3	4.5109	95	68
25	2	24.3	95	162	98.6	54	3	3.8501	87	49
25	1	26	92	187	120.4	56	3	3.9703	88	68
61	2	32	103.67	210	85.2	35	6	6.107	124	245
31	1	29.7	88	167	103.4	48	4	4.3567	78	184
30	2	25.2	83	178	118.4	34	5	4.852	83	202
19	1	19.2	87	124	54	57	2	4.1744	90	137

This figure captures only the top part of the data. On the webpage you need to scroll down considerably to view the whole content. Thus, to get an **overview** of the dataset is the first main task in Data Science.

3 The lesson

- introduces code to read and inspect the data
- works with a specific data frame and extracts some techniques to get an overview
- discusses the concept 'distribution' as a way of summarising data in a single figure

3.1 To get to know a dataset you need to

- access the data
- check the content
- produce a summary of basic properties

In this lesson we will only look at univariate features where each data column is studied independently of the others. Further properties and bivariate features will be the topic of the next lesson.

4 Work Through Example

4.1 Reading data into a Pandas DataFrame

The small practice data file for this section is called 'everleys_data.csv' and can be downloaded using the link given above in "Materials for this Lesson". To start, please create a subfolder called 'data' in the current directory and put the data file in it. It can now be accessed using the relative path `data/everleys_data.csv` or `data\everleys_data.csv`, respectively.

The file `everleys_data.csv` contains blood concentrations of calcium and sodium ions from 17 patients with Everley's syndrome. The data are taken from a BMJ statistics tutorial. The data are stored as comma-separated values (csv), two values for each patient.

To get to know a dataset, we will use the Pandas package and the Matplotlib plotting library. The Pandas package for data science is included in the Anaconda distribution of Python. Check this link for installation instructions to get started.

If you are not using the Anaconda distribution, please refer to these guidelines.

To use the functions contained in Pandas they need to be imported. Our dataset is in '.csv' format, and we therefore need to read it from a csv file. For this, we import the function `read_csv`. This function will create a *Pandas dataframe*.

[1]:
```
from pandas import read_csv
```

Executing this code does not lead to any output on the screen. However, the function is now ready to be used. To use it, we type its name and provide the required arguments. The following code should import the Everley's data into your JupyterLab notebook (or other Python environment):

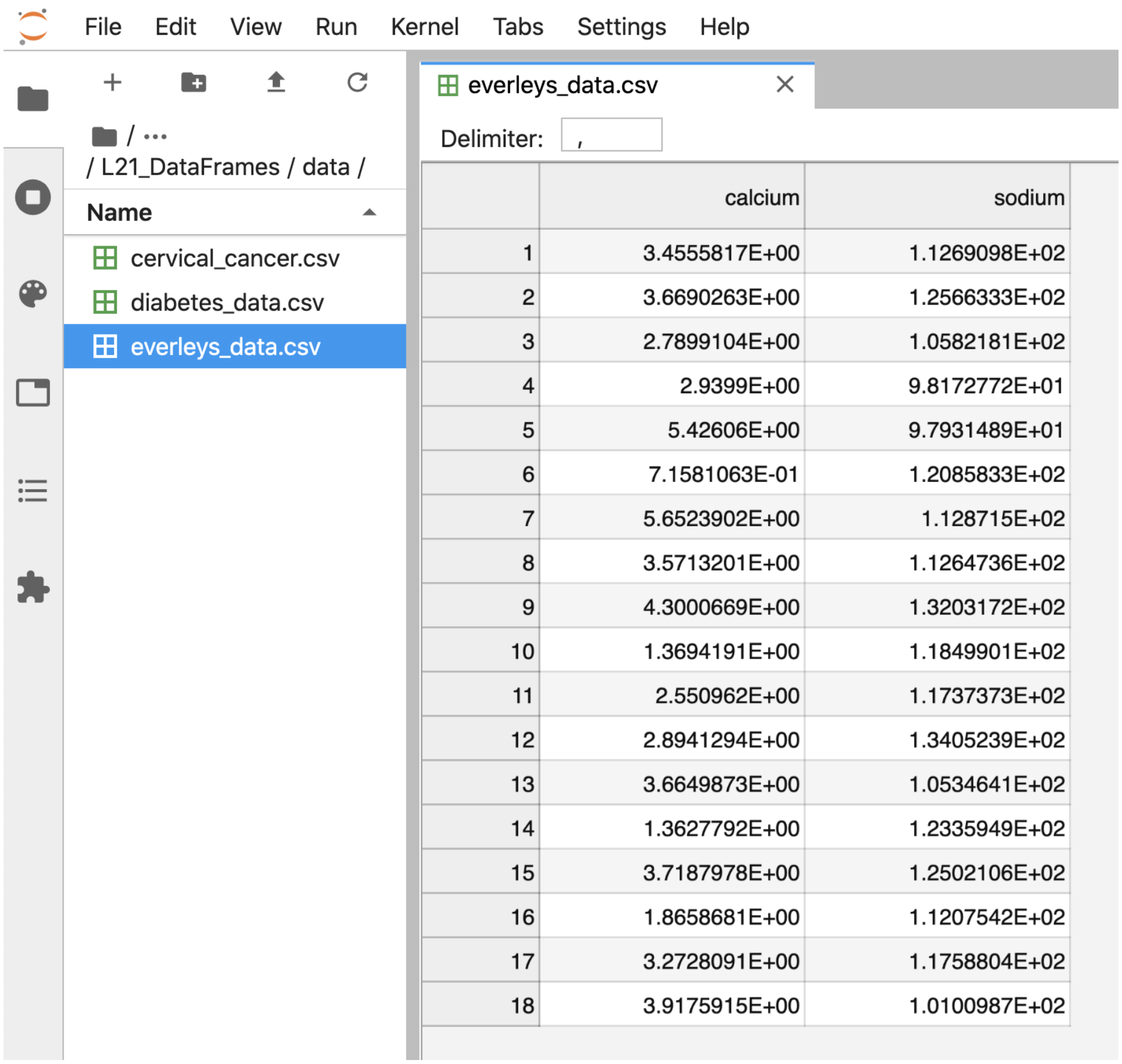
[2]:
```
# for Mac OSX and Linux
# (please go to the next cell if using Windows)

df = read_csv("data/everleys_data.csv")
```

[3]:
```
# Please uncomment for Windows
# (please go to previous cell if using Mac OSX or Linux)

# df = read_csv("data\everleys_data.csv")
```

This code uses the `read_csv` function from Pandas to read data from a data file, in this case a file with extension '.csv'. Note that the location of the data file is specified within quotes by the relative path to the subfolder ‘data’ followed by the file name. Use the JupyterLab file browser to check that subfolder exists and has the file in it.



	calcium	sodium
1	3.4555817E+00	1.1269098E+02
2	3.6690263E+00	1.2566333E+02
3	2.7899104E+00	1.0582181E+02
4	2.9399E+00	9.8172772E+01
5	5.42606E+00	9.7931489E+01
6	7.1581063E-01	1.2085833E+02
7	5.6523902E+00	1.128715E+02
8	3.5713201E+00	1.1264736E+02
9	4.3000669E+00	1.3203172E+02
10	1.3694191E+00	1.1849901E+02
11	2.550962E+00	1.1737373E+02
12	2.8941294E+00	1.3405239E+02
13	3.6649873E+00	1.0534641E+02
14	1.3627792E+00	1.2335949E+02
15	3.7187978E+00	1.2502106E+02
16	1.8658681E+00	1.1207542E+02
17	3.2728091E+00	1.1758804E+02
18	3.9175915E+00	1.0100987E+02

After execution of the code, the data are contained in a variable called `df`. This is a structure referred to as a Pandas *DataFrame*.

> A **Pandas dataframe** is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it as a spreadsheet.

To see the contents of `df`, simply use:

[4]:
```
df
```

[4]:
```
   calcium      sodium
0  3.455582  112.690980
1  3.669026  125.663330
2  2.789910  105.821810
3  2.939900   98.172772
4  5.426060   97.931489
```

```
5   0.715811  120.858330
6   5.652390  112.871500
7   3.571320  112.647360
8   4.300067  132.031720
9   1.369419  118.499010
10  2.550962  117.373730
11  2.894129  134.052390
12  3.664987  105.346410
13  1.362779  123.359490
14  3.718798  125.021060
15  1.865868  112.075420
16  3.272809  117.588040
17  3.917591  101.009870
```

	calcium	sodium
0	3.455582	112.690980
1	3.669026	125.663330
2	2.789910	105.821810
3	2.939900	98.172772
4	5.426060	97.931489
5	0.715811	120.858330
6	5.652390	112.871500
7	3.571320	112.647360
8	4.300067	132.031720
9	1.369419	118.499010
10	2.550962	117.373730
11	2.894129	134.052390
12	3.664987	105.346410
13	1.362779	123.359490
14	3.718798	125.021060
15	1.865868	112.075420
16	3.272809	117.588040
17	3.917591	101.009870

(Compare with the result of `print(df)` which displays the contents in a different format.)

The output shows in the first column an index, integers from 0 to 17; and the calcium and sodium concentrations in columns 2 and 3, respectively. The default indexing starts from zero (Python is a 'zero-based' programming language).

In a dataframe, the first column is referred to as *Indices*, the first row is referred to as *Labels*. Note that the row with the labels is excluded from the row count. Similarly, the row with the indices is excluded from the column count.

For large data sets, the function `head` is a convenient way to get a feel of the dataset.

[5]:
```
df.head()
```

[5]:

	calcium	sodium
0	3.455582	112.690980
1	3.669026	125.663330
2	2.789910	105.821810
3	2.939900	98.172772
4	5.426060	97.931489


```
    calcium      sodium
0  3.455582  112.690980
1  3.669026  125.663330
2  2.789910  105.821810
3  2.939900   98.172772
4  5.426060   97.931489
```


Without any input argument, this displays the first five data lines of the dataframe. You can specify alter the number of rows displayed by including a single integer as argument, e.g. `head(10)`.

If you feel there are too many decimal places in the default view, you can restrict their number by using the `round` function:

[6]:
```
df.head().round(2)
```

[6]:

	calcium	sodium
0	3.46	112.69
1	3.67	125.66
2	2.79	105.82
3	2.94	98.17
4	5.43	97.93


```
   calcium  sodium
0     3.46  112.69
1     3.67  125.66
2     2.79  105.82
3     2.94   98.17
4     5.43   97.93
```


While we can see how many rows there are in a dataframe when we display the whole data frame and look at the last index, there is a convenient way to obtain the number directly:

[7]:
```
no_rows = len(df)

print('Data frame has', no_rows, 'rows')
```

```
Data frame has 18 rows
```


```
Data frame has 18 rows
```


You could see above, that the columns of the dataframe have labels. To see all labels:

[8]:
```
column_labels = df.columns

print(column_labels)
```

```
Index(['calcium', 'sodium'], dtype='object')
```


```
Index(['calcium', 'sodium'], dtype='object')
```


Now we can count the labels to obtain the number of columns:

[9]:
```
no_columns = len(column_labels)

print('Data frame has', no_columns, 'columns')
```

```
Data frame has 2 columns
```


```
Data frame has 2 columns
```


And if you want to have both the number of the rows and the columns together, use `shape`. Shape returns a tuple of two numbers, first the number of rows, then the number of columns.

[10]:
```
df_shape = df.shape

print('Data frame has', df_shape[0], 'rows and',df_shape[1],  'columns')
```

```
Data frame has 18 rows and 2 columns
```


```
Data frame has 18 rows and 2 columns
```


Notice that `shape` (like `columns`) is not followed by round parenthesis. It is not a function that can take arguments. Technically, `shape` is a 'property' of the dataframe.

To find out what data type is contained in each of the columns, us `dtypes`, another 'property':

[11]:
```
df.dtypes
```

[11]:
```
calcium    float64
sodium     float64
dtype: object
```


```
calcium    float64
sodium     float64
dtype: object
```


In this case, both columns contain floating point (decimal) numbers.

DIY1: Read data into a dataframe

Download the data file 'loan_data.csv' using the link given above in "Materials for this Lesson". It contains data that can be used for the assessment of loan applications. Read the data into a DataFrame. It is best to assign it a name other than 'df' (to avoid overwriting the Evereley data set).

Display the first ten rows of the Loan data set to see its contents. It is taken from a tutorial on Data Handling in Python which you might find useful for further practice.

> From this exercise we can see that a dataframe can contain different types of data: real numbers (e.g. LoanAmount), integers (ApplicantIncome), categorical data (Gender), and strings (Loan_ID).

[25]:
```python
from pandas import read_csv
# dataframe from .csv file
df_loan = read_csv("data/loan_data.csv")
# display contents
df_loan.head(10)
```

[25]:
```
    Loan_ID Gender Married Dependents     Education Self_Employed  \
0  LP001015   Male     Yes          0      Graduate            No
1  LP001022   Male     Yes          1      Graduate            No
2  LP001031   Male     Yes          2      Graduate            No
3  LP001035   Male     Yes          2      Graduate            No
4  LP001051   Male      No          0  Not Graduate            No
5  LP001054   Male     Yes          0  Not Graduate           Yes
6  LP001055 Female      No          1  Not Graduate            No
7  LP001056   Male     Yes          2  Not Graduate            No
8  LP001059   Male     Yes          2      Graduate           NaN
9  LP001067   Male      No          0  Not Graduate            No

   ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  \
0             5720                  0       110.0             360.0
1             3076               1500       126.0             360.0
2             5000               1800       208.0             360.0
3             2340               2546       100.0             360.0
4             3276                  0        78.0             360.0
5             2165               3422       152.0             360.0
6             2226                  0        59.0             360.0
7             3881                  0       147.0             360.0
8            13633                  0       280.0             240.0
9             2400               2400       123.0             360.0

   Credit_History Property_Area
0             1.0         Urban
1             1.0         Urban
2             1.0         Urban
3             NaN         Urban
4             1.0         Urban
5             1.0         Urban
6             1.0     Semiurban
7             0.0         Rural
8             1.0         Urban
9             1.0     Semiurban
```

```
    Loan_ID  Gender Married  ... Loan_Amount_Term Credit_History Property_Area
0  LP001015    Male     Yes  ...            360.0            1.0         Urban
1  LP001022    Male     Yes  ...            360.0            1.0         Urban
2  LP001031    Male     Yes  ...            360.0            1.0         Urban
3  LP001035    Male     Yes  ...            360.0            NaN         Urban
4  LP001051    Male      No  ...            360.0            1.0         Urban
5  LP001054    Male     Yes  ...            360.0            1.0         Urban
6  LP001055  Female      No  ...            360.0            1.0     Semiurban
7  LP001056    Male     Yes  ...            360.0            0.0         Rural
8  LP001059    Male     Yes  ...            240.0            1.0         Urban
9  LP001067    Male      No  ...            360.0            1.0     Semiurban

[10 rows x 12 columns]
```

5 Accessing data in a DataFrame

If a datafile is large and you only want to check the format of data in a specific column, you can limit the display to that column. To access data contained in a specific column of a dataframe, we can use a similar convention as in a Python dictionary, treating the column names as 'keys'. E.g. to show all rows in column 'Calcium', use:

[26]:
```
df['calcium']
```

[26]:
```
0     3.455582
1     3.669026
2     2.789910
3     2.939900
4     5.426060
5     0.715811
6     5.652390
7     3.571320
8     4.300067
9     1.369419
10    2.550962
11    2.894129
12    3.664987
13    1.362779
14    3.718798
15    1.865868
16    3.272809
17    3.917591
Name: calcium, dtype: float64
```

```
0     3.455582
1     3.669026
2     2.789910
3     2.939900
```

```
4     5.426060
5     0.715811
6     5.652390
7     3.571320
8     4.300067
9     1.369419
10    2.550962
11    2.894129
12    3.664987
13    1.362779
14    3.718798
15    1.865868
16    3.272809
17    3.917591
Name: calcium, dtype: float64
```

To access individual rows of a column we use two pairs of square brackets:

[27]:
```python
df['calcium'][0:3]
```

[27]:
```
0    3.455582
1    3.669026
2    2.789910
Name: calcium, dtype: float64
```


```
0    3.455582
1    3.669026
2    2.789910
Name: calcium, dtype: float64
```


Here all rules for slicing can be applied. As for lists and tuples, the indexing of rows is semi-inclusive, lower boundary included, upper boundary excluded. Note that the first pair of square brackets refers to a column and the second pair refers to the rows. This is different from e.g. accessing items in a nested list.

Accessing items in a Pandas dataframe is analogous to accessing the values in a Python dictionary by referring to its keys.

To access non-contiguous elements, we use an additional pair of square brackets (as if for a list within a list):

[28]:
```python
df['calcium'][[1, 3, 7]]
```

[28]:
```
1    3.669026
3    2.939900
7    3.571320
Name: calcium, dtype: float64
```


```
1    3.669026
3    2.939900
```

```
7    3.571320
Name: calcium, dtype: float64
```

Another possibility to index and slice a dataframe is the use of the 'index location' or `iloc` property. It refers first to rows and then to columns by index, all within a single pair of brackets. For example, to get all rows of the first column (index `0`), you use:

[29]:
```python
df.iloc[:, 0]
```

[29]:
```
0     3.455582
1     3.669026
2     2.789910
3     2.939900
4     5.426060
5     0.715811
6     5.652390
7     3.571320
8     4.300067
9     1.369419
10    2.550962
11    2.894129
12    3.664987
13    1.362779
14    3.718798
15    1.865868
16    3.272809
17    3.917591
Name: calcium, dtype: float64
```


```
0     3.455582
1     3.669026
2     2.789910
3     2.939900
4     5.426060
5     0.715811
6     5.652390
7     3.571320
8     4.300067
9     1.369419
10    2.550962
11    2.894129
12    3.664987
13    1.362779
14    3.718798
15    1.865868
16    3.272809
17    3.917591
Name: calcium, dtype: float64
```

To display only the first three calcium concentrations, you use slicing, remembering that the upper bound is excluded):

[30]:
```
df.iloc[0:3, 0]
```

[30]:
```
0    3.455582
1    3.669026
2    2.789910
Name: calcium, dtype: float64
```


```
0    3.455582
1    3.669026
2    2.789910
Name: calcium, dtype: float64
```


To access non-consecutive values, we can use a pair of square brackets within the pair of square brackets:

[31]:
```
df.iloc[[2, 4, 7], 0]
```

[31]:
```
2    2.78991
4    5.42606
7    3.57132
Name: calcium, dtype: float64
```


```
2    2.78991
4    5.42606
7    3.57132
Name: calcium, dtype: float64
```


Similarly, we can access the values from multiple columns:

[32]:
```
df.iloc[[2, 4, 7], :]
```

[32]:
```
   calcium      sodium
2  2.78991  105.821810
4  5.42606   97.931489
7  3.57132  112.647360
```


```
   calcium      sodium
2  2.78991  105.821810
4  5.42606   97.931489
7  3.57132  112.647360
```


To pick only the even rows from the two columns, check this colon notation:

[33]:
```
df.iloc[:18:2, :]
```

[33]:
```
    calcium      sodium
0  3.455582  112.690980
```

```
2    2.789910  105.821810
4    5.426060   97.931489
6    5.652390  112.871500
8    4.300067  132.031720
10   2.550962  117.373730
12   3.664987  105.346410
14   3.718798  125.021060
16   3.272809  117.588040
```

```
     calcium      sodium
0    3.455582  112.690980
2    2.789910  105.821810
4    5.426060   97.931489
6    5.652390  112.871500
8    4.300067  132.031720
10   2.550962  117.373730
12   3.664987  105.346410
14   3.718798  125.021060
16   3.272809  117.588040
```

The number after the second colon indicates the stepsize.

DIY2: Select data from dataframe

Display the calcium and sodium concentrations of all patients except the first. Check the model solution at the bottom for options.

[34]:
```
df[['calcium', 'sodium']][1:]
```

[34]:
```
     calcium      sodium
1    3.669026  125.663330
2    2.789910  105.821810
3    2.939900   98.172772
4    5.426060   97.931489
5    0.715811  120.858330
6    5.652390  112.871500
7    3.571320  112.647360
8    4.300067  132.031720
9    1.369419  118.499010
10   2.550962  117.373730
11   2.894129  134.052390
12   3.664987  105.346410
13   1.362779  123.359490
14   3.718798  125.021060
15   1.865868  112.075420
16   3.272809  117.588040
17   3.917591  101.009870
```

```
    calcium      sodium
1   3.669026  125.663330
2   2.789910  105.821810
3   2.939900   98.172772
4   5.426060   97.931489
5   0.715811  120.858330
6   5.652390  112.871500
7   3.571320  112.647360
8   4.300067  132.031720
9   1.369419  118.499010
10  2.550962  117.373730
11  2.894129  134.052390
12  3.664987  105.346410
13  1.362779  123.359490
14  3.718798  125.021060
15  1.865868  112.075420
16  3.272809  117.588040
17  3.917591  101.009870
```

Mixing the ways to access specific data in a dataframe can be confusing and needs practice.

5.1 Search for missing values

Some tables contain missing entries. You can check a dataframe for such missing entries. If no missing entry is found, the function `isnull` will return `False`.

[35]:
```
df.isnull().any()
```

[35]:
```
calcium    False
sodium     False
dtype: bool
```

```
calcium    False
sodium     False
dtype: bool
```

This shows that there are no missing entries in our dataframe.

DIY3: Find NaN in dataframe

In the Loan data set, check the entry 'Self-employed' for ID LP001059. It shows how a missing value is represented as 'NaN' (not a number).

Verify that the output of `isnull` in this case is `True`

[36]:
```
df_loan['Self_Employed'][8]
```

[36]:
```
nan
```

```
nan
```

[37]:
```python
df_loan['Self_Employed'][8:9].isnull()
```

[37]:
```
8    True
Name: Self_Employed, dtype: bool
```


```
8    True
Name: Self_Employed, dtype: bool
```


6 Basic data features: Summary Statistics

To get a summary of basic data features use the function `describe`:

[38]:
```python
description = df.describe()

description
```

[38]:

	calcium	sodium
count	18.000000	18.000000
mean	3.174301	115.167484
std	1.306652	10.756852
min	0.715811	97.931489
25%	2.610699	107.385212
50%	3.364195	115.122615
75%	3.706355	122.734200
max	5.652390	134.052390


```
         calcium      sodium
count  18.000000   18.000000
mean    3.174301  115.167484
std     1.306652   10.756852
min     0.715811   97.931489
25%     2.610699  107.385212
50%     3.364195  115.122615
75%     3.706355  122.734200
max     5.652390  134.052390
```


The `describe` function produces a new dataframe (here called 'description') that contains the number of samples, the mean, the standard deviation, minimum, 25th, 50th, 75th percentile, and the maximum value for each column of the data. Note that the indices of the rows have now been replaced by strings. To access rows, it is possible to refer to those names using the `loc` property. E.g. to access the mean of the calcium concentrations from the description, each of the following is valid:

[39]:
```python
# Option 1
description.loc['mean']['calcium']
```

```python
# Option 2
description.loc['mean'][0]

# Option 3
description['calcium']['mean']

# Option 4
description['calcium'][1]
```

[39]: 3.1743005405555547

```
3.1743005405555555
3.1743005405555555
3.1743005405555555
3.1743005405555555
```

DIY4: Practice

Use your own .csv data set to practice. (If you don't have a data set at hand, any excel table can be exported as .csv.) Read it into a dataframe, check its header, access individual values or sets of values. Create a statistics using `describe` and check for missing values using `.isnull`.

[ad libitum]

Iterating through the columns

Now we know how to access all data in a dataframe and how to get a summary statistics over each column.

Here is code to iterate through the columns and access the first two concentrations:

[40]:
```python
for col in df:

    print(df[col][0:2])
```

```
0    3.455582
1    3.669026
Name: calcium, dtype: float64
0    112.69098
1    125.66333
Name: sodium, dtype: float64
```


```
0    3.455582
1    3.669026
Name: calcium, dtype: float64
0    112.69098
1    125.66333
Name: sodium, dtype: float64
```

As a slightly more complex example, we access the median ('50%') of each column in the description and add it to a list:

[41]:
```
conc_medians = list()

for col in df:

    conc_medians.append(df[col].describe()['50%'])

print('The columns are: ', list(df.columns))
print('The medians are: ', conc_medians)
```

```
The columns are:  ['calcium', 'sodium']
The medians are:  [3.3641954, 115.122615]
```


```
The columns are:  ['calcium', 'sodium']
The medians are:  [3.3641954, 115.122615]
```


This approach is useful for data frames with a large number of columns. For instance, it is possible to then create a boxplot or histogram for the means, medians etc. of the dataframe and thus to get an overview of all (comparable) columns.

Selecting a subset based on a template

An analysis of a data set may need to be done on part of the data. This can often be formulated by using a logical condition which specifies the required subset.

For this we will assume that some of the data are labelled '0' and some are labelled '1'. Let us therefore see how to add a new column to our Evereleys data frame which contains the (in this case arbitrary) labels.

First we randomly create as many labels as we have rows in the data frame. We can use the `randint` function which we import from 'numpy.random'. `randint` in its simple form takes two arguments. First the upper bound of the integer needed, where by default it starts from zero. As Python is exclusive on the upper bound, providing '2' will thus yield either '0' or '1' only.

[42]:
```
from numpy.random import randint

no_rows = len(df)

randomLabel = randint(2, size=no_rows)

print('Number of rows:  ', no_rows)
print('Number of Labels:', len(randomLabel))
print('Labels:          ', randomLabel)
```

```
Number of rows:   18
Number of Labels: 18
Labels:           [0 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1]
```


```
Number of rows:   18
```

```
Number of Labels: 18
Labels:            [1 0 1 1 1 1 1 0 1 1 0 1 1 1 0 1 0 0]
```

Note how we obtain the number of rows (18) using `len` and do not put it directly into the code.

Now we create a new data column in our `df` dataframe which contains the labels. To create a new column, you can simply refer to a column name that does not yet exist and assign values to it. Let us call it 'gender', assuming that '0' represents male and '1' represents female.

As gender specification can include more than two labels, try to create a column with more than two randomly assigned labels e.g. (0, 1, 2).

[43]:
```
df['gender'] = randomLabel

df.head()
```

[43]:

	calcium	sodium	gender
0	3.455582	112.690980	0
1	3.669026	125.663330	1
2	2.789910	105.821810	1
3	2.939900	98.172772	0
4	5.426060	97.931489	1

	calcium	sodium	gender
0	3.455582	112.690980	1
1	3.669026	125.663330	0
2	2.789910	105.821810	1
3	2.939900	98.172772	1
4	5.426060	97.931489	1

Now we can use the information contained in 'gender' to filter the data by gender. To achieve this, we use a conditional statement. Let us check which of the rows are labelled as '1':

[44]:
```
df['gender'] == 1
```

[44]:
```
0     False
1      True
2      True
3     False
4      True
5      True
6     False
7     False
8      True
9      True
10    False
11    False
12     True
13    False
14     True
```

```
15     True
16    False
17     True
Name: gender, dtype: bool

0      True
1     False
2      True
3      True
4      True
5      True
6      True
7     False
8      True
9      True
10    False
11     True
12     True
13     True
14    False
15     True
16    False
17    False
Name: gender, dtype: bool
```

If we assign the result of the conditional statement (Boolean True or False) to a variable, then this variable can act as a template to filter the data. If we call the data frame with that variable, we will only get the rows where the condition was found to be True:

[45]:
```
df_female = df['gender'] == 1

df[df_female]
```

[45]:
```
     calcium      sodium  gender
1   3.669026  125.663330       1
2   2.789910  105.821810       1
4   5.426060   97.931489       1
5   0.715811  120.858330       1
8   4.300067  132.031720       1
9   1.369419  118.499010       1
12  3.664987  105.346410       1
14  3.718798  125.021060       1
15  1.865868  112.075420       1
17  3.917591  101.009870       1

    calcium      sodium  gender
0  3.455582  112.690980       1
2  2.789910  105.821810       1
```

```
3    2.939900   98.172772       1
4    5.426060   97.931489       1
5    0.715811  120.858330       1
6    5.652390  112.871500       1
8    4.300067  132.031720       1
9    1.369419  118.499010       1
11   2.894129  134.052390       1
12   3.664987  105.346410       1
13   1.362779  123.359490       1
15   1.865868  112.075420       1
```

Using the Boolean, we only pick the rows that are labelled '1' and thus get a subset of the data according to the label.

DIY5: Using a template

Modify the code to calculate the number of samples labelled 0 and check the number of rows of that subset.

[46]:
```
from numpy.random import randint
no_rows = len(df)
randomLabel = randint(2, size=no_rows)
df['gender'] = randomLabel
df_male = df['gender'] == 0
no_males = len(df[df_male])
print(no_males, 'samples are labelled "male".')
```

```
9 samples are labelled "male".
```

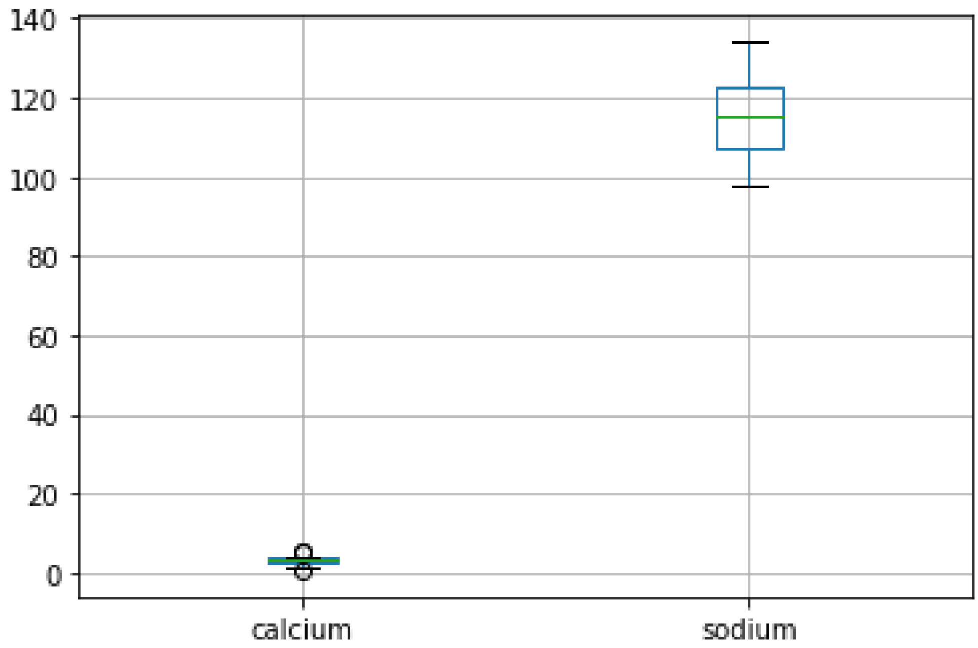
```
11 samples are labelled "male".
```

7 Visualisation of data

It is easy to see from the numbers that the concentrations of sodium are much higher than that of calcium. However, to also compare the medians, percentiles and the spread of the data it is better to use visualisation.

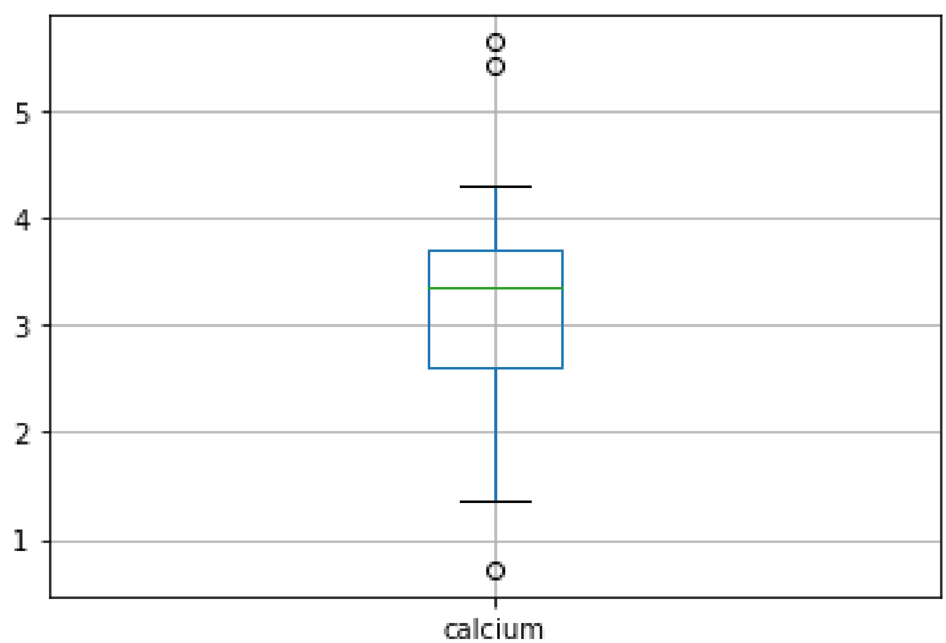
The simplest way of visualisation is to use Pandas functionality which offers direct ways of plotting. Here is an example where a boxplot is created for each column:

[48]:
```
df = read_csv("data/everleys_data.csv")
df.boxplot();
```



By default, boxplots are shown for all columns if no further argument is given to the function (empty round parenthesis). As the calcium plot is rather squeezed we may wish to see it individually. This can be done by specifying the calcium column as an argument:

[50]:
```
# Boxplot of calcium results
df.boxplot(column='calcium');
```



Using Matplotlib

> **Matplotlib** is a comprehensive library for creating static, animated, and interactive visualizations in Python.

The above is an easy way to create boxplots directly on the dataframe. It is based on the library Matplotlib and specifically uses the **pyplot library**. For simplicity, the code is put in a convenient Pandas function.

However, we are going to use **Matplotlib** extensively later on in the course, and we therefore now introduce the direct, generic way of using it.

For this, we import the function `subplots` from the pyplot library:

[51]:
```
from matplotlib.pyplot import subplots, show
```

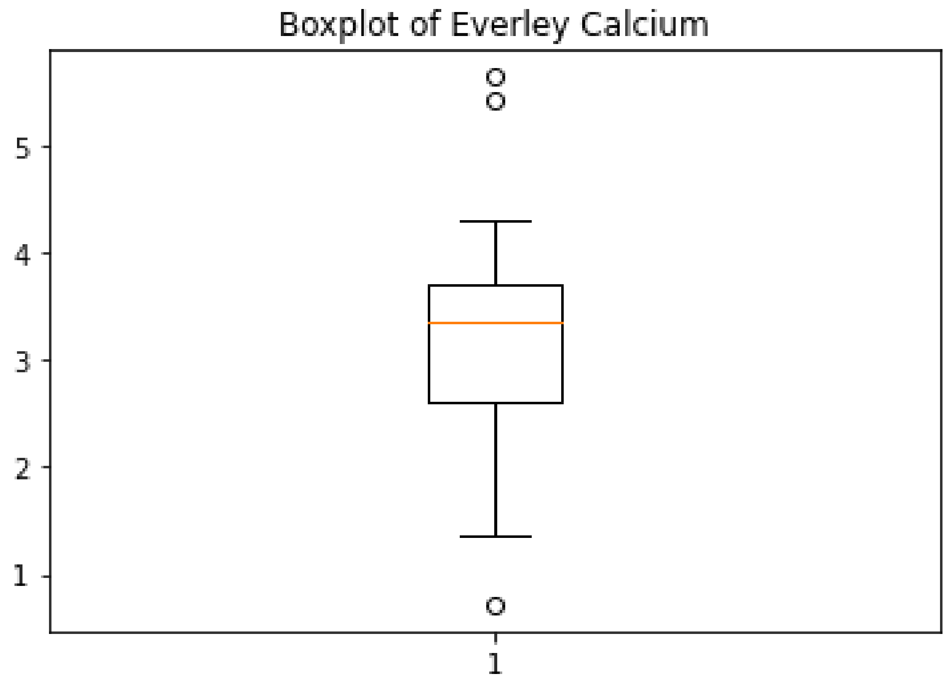
The way to use `subplots` is to first set up a figure environment (below it is called 'fig') and an empty coordinate system (below called 'ax'). The plot is then done using one of the many methods available in Matplotlib. We apply it to the coordinate system 'ax'.

As an example, let us create a boxplot of the calcium variable. As an argument of the function we need to specify the data. We can use the values of the 'calcium' concentrations from the column with that name:

[52]:
```
fig, ax = subplots()

ax.boxplot(df['calcium'])
ax.set_title('Boxplot of Everley Calcium')

show()
```



Note how following the actual plot we define the title of the plot by referring to the same coordinate system `ax`.

The value of `subplots` becomes apparent when we try to create more than one plot in a single figure.

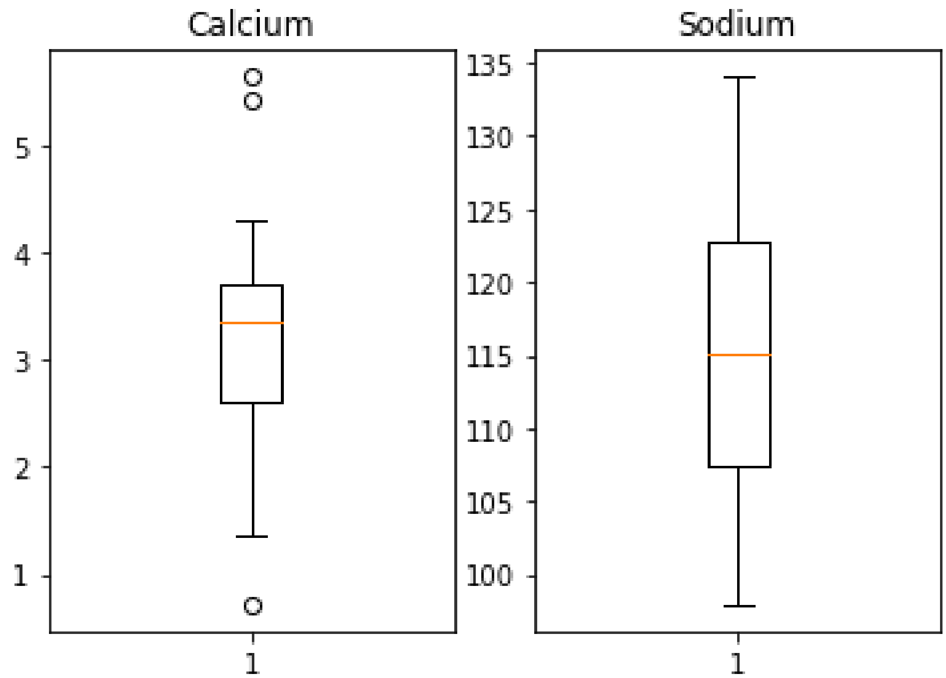
Here is an example to create two boxplots next to each other. The keyword arguments to use is 'ncols' which is the number of figures per row. 'ncols=2' indicates that you want to have two plots next to each other.

[53]:
```
fig, ax = subplots(ncols=2)

ax[0].boxplot(df['calcium'])
ax[0].set_title('Calcium')

ax[1].boxplot(df['sodium'])
ax[1].set_title('Sodium');

show()
```



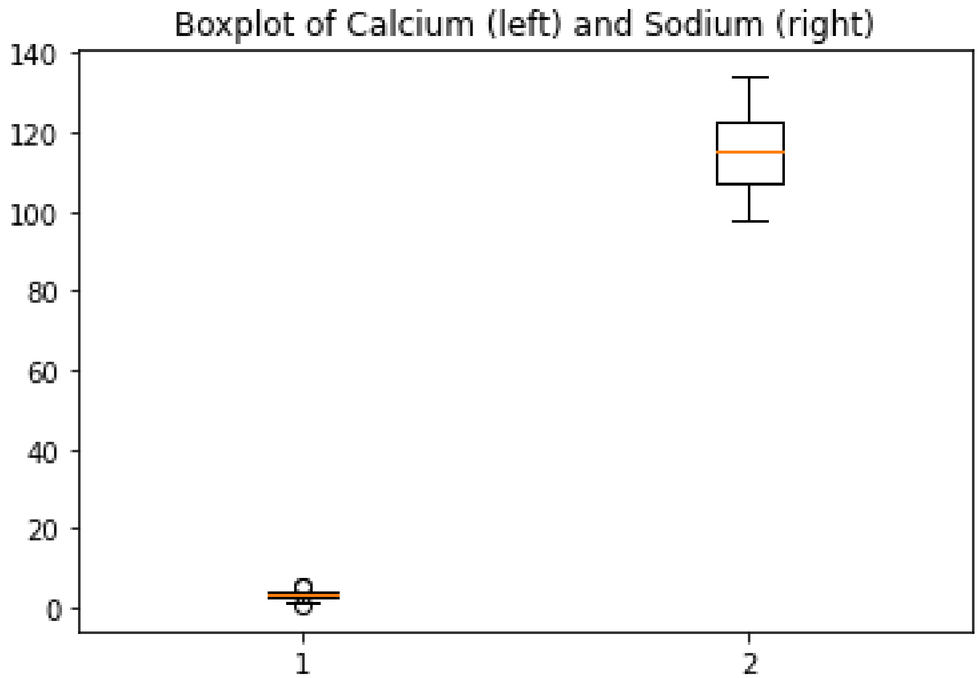
Note that you now have to refer to each of the subplots by indexing the coordinate system 'ax'. This figure gives a good overview of the Everley's data.

If you prefer to have the boxplots of both columns in a single figure, that can also be done:

[54]:
```
fig, ax = subplots(ncols=1, nrows=1)

ax.boxplot([df['calcium'], df['sodium']], positions=[1, 2])
ax.set_title('Boxplot of Calcium (left) and Sodium (right)')

show()
```

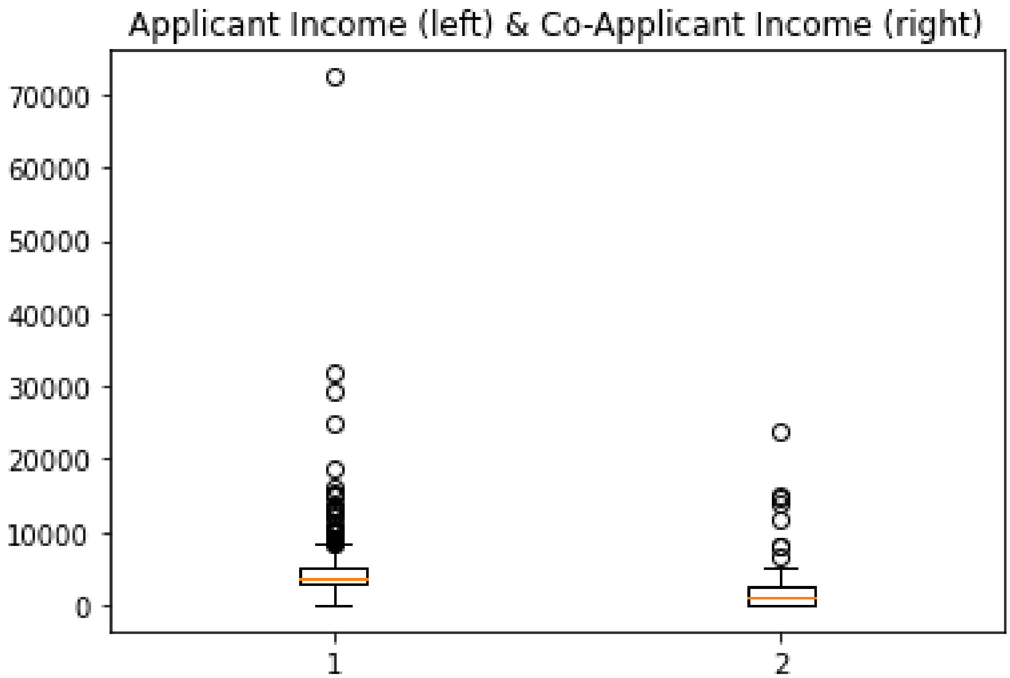


DIY6: Boxplot from Loan data

Plot the boxplots of the 'ApplicantIncome' and the 'CoapplicantIncome' in the Loan data using the above code.

[55]:
```
fig, ax = subplots(ncols=1, nrows=1)
ax.boxplot([df_loan['ApplicantIncome'], df_loan['CoapplicantIncome']],␣
↪positions=[1, 2])
ax.set_title('Applicant Income (left) & Co-Applicant Income (right)');

show()
```



7.1 Histogram

Another good overview is the histogram: Containers or 'bins' are created over the range of values found within a column and the count of the values for each bin is plotted on the vertical axis.

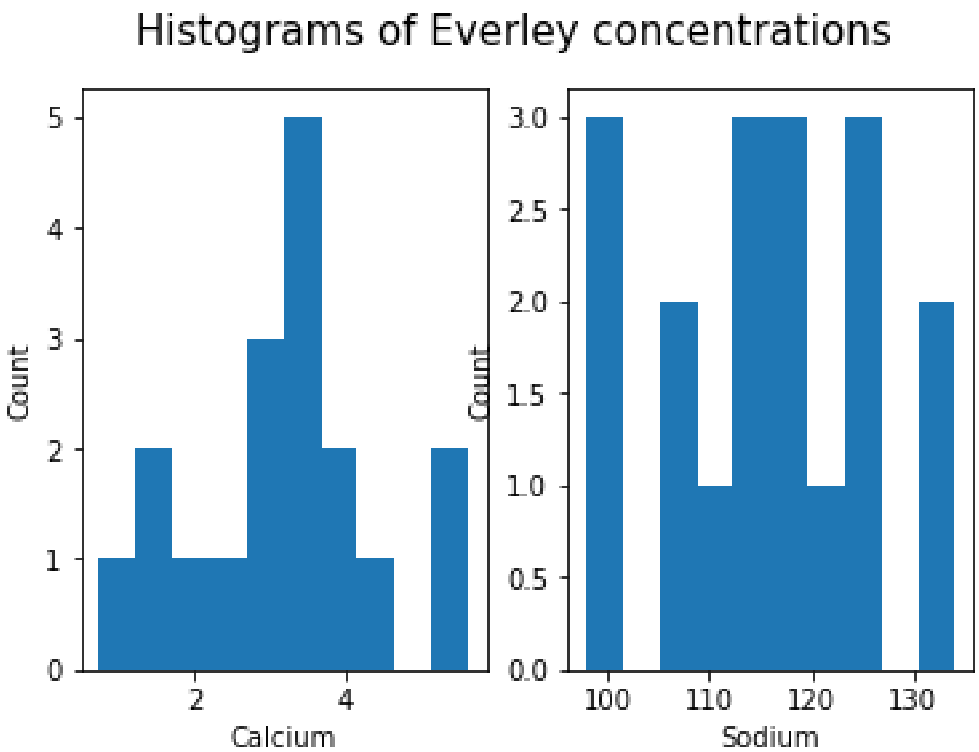
[56]:
```
fig, ax = subplots(ncols=2, nrows=1)

ax[0].hist(df['calcium'])
ax[0].set(xlabel='Calcium', ylabel='Count');

ax[1].hist(df['sodium'])
ax[1].set(xlabel='Sodium', ylabel='Count');

fig.suptitle('Histograms of Everley concentrations', fontsize=15);

show()
```



This also shows how to add labels to the axes and a title to the overall figure.

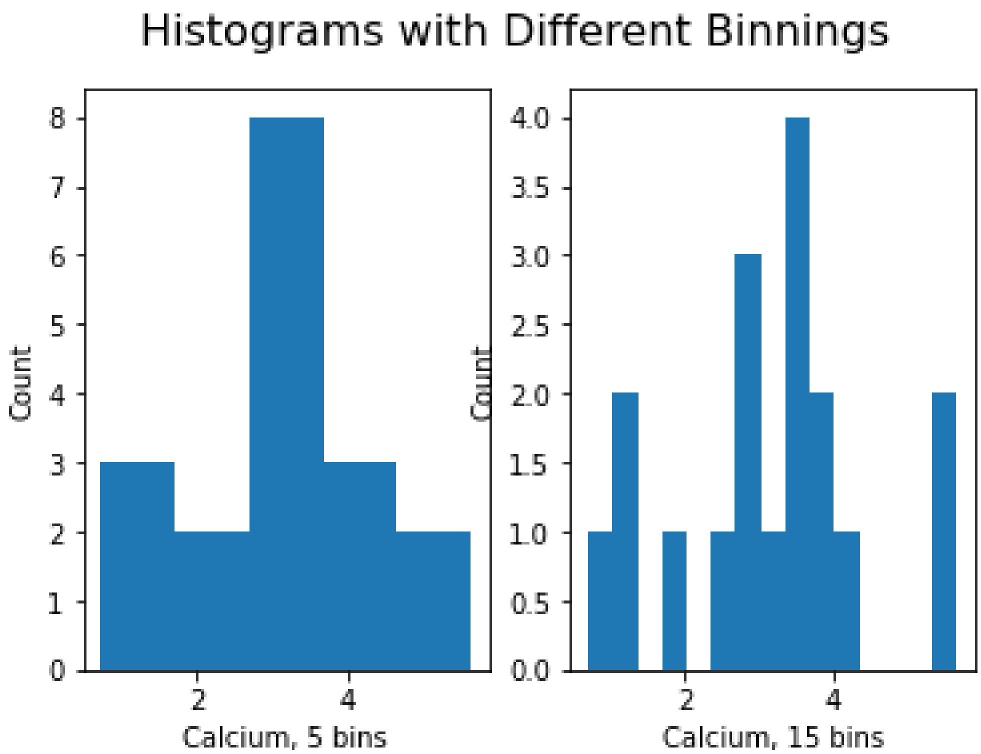
This uses the default value for the generation of the bins. It is set to 10 bins over the range of which values are found. The number of bins in the histogram can be changed using the keyword argument 'bins':

[57]:
```
fig, ax = subplots(ncols=2, nrows=1)

ax[0].hist(df['calcium'], bins=5)
ax[0].set(xlabel='Calcium, 5 bins', ylabel='Count');

ax[1].hist(df['calcium'], bins=15)
ax[1].set(xlabel='Calcium, 15 bins', ylabel='Count');
fig.suptitle('Histograms with Different Binnings', fontsize=16);

show()
```



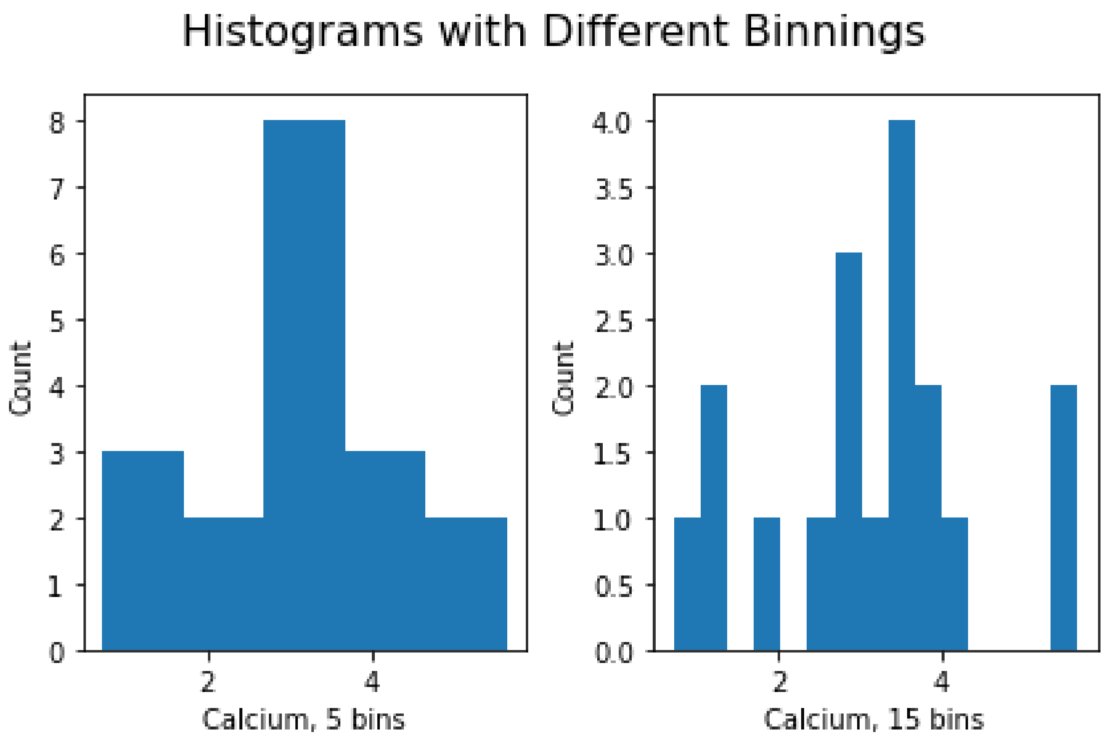
Note how the y-label of the right figure is not placed well. To correct for the placement of the labels and the title, you can use `tight_layout` on the figure:

[58]:
```
fig, ax = subplots(ncols=2, nrows=1)

ax[0].hist(df['calcium'], bins=5)
ax[0].set(xlabel='Calcium, 5 bins', ylabel='Count');

ax[1].hist(df['calcium'], bins=15)
ax[1].set(xlabel='Calcium, 15 bins', ylabel='Count');
fig.suptitle('Histograms with Different Binnings', fontsize=16);
fig.tight_layout()

show()
```

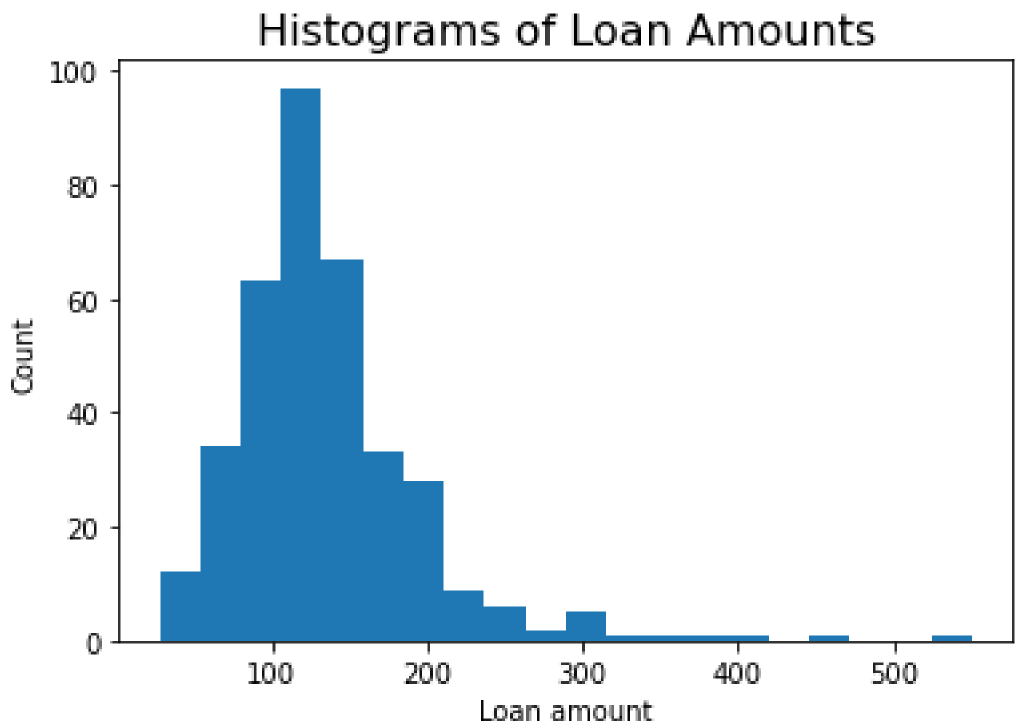


DIY7: Create the histogram of a column

Take the loan data and display the histogram of the loan amount that people asked for. (Loan amounts are divided by 1000, i.e. in k£ on the horizontal axis). Use e.g. 20 bins.

[60]:
```
# Histogram of loan amounts in k£
fig, ax = subplots()
ax.hist(df_loan['LoanAmount'], bins=20)
ax.set(xlabel='Loan amount', ylabel='Count');
ax.set_title('Histograms of Loan Amounts', fontsize=16);

show()
```



8 Handling the Diabetes Data Set

We now return to the data set that started our enquiry into the handling of data in a dataframe.

We will now:

- Import the diabetes data from 'sklearn'
- Check the shape of the dataframe and search for NANs
- Get a summary plot of one of its statistical quantities (e.g. mean) for all columns

First we import the data set and check its `head`. Wait until the numbers show below the code, it might take a while.

[61]:
```
from sklearn import datasets

diabetes = datasets.load_diabetes()

X = diabetes.data

from pandas import DataFrame

df_diabetes = DataFrame(data=X)

df_diabetes.head()
```

[61]:
```
          0         1         2         3         4         5         6  \
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
2  0.085299  0.050680  0.044451 -0.005671 -0.045599 -0.034194 -0.032356
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

          7         8         9
0 -0.002592  0.019908 -0.017646
1 -0.039493 -0.068330 -0.092204
2 -0.002592  0.002864 -0.025930
3  0.034309  0.022692 -0.009362
4 -0.002592 -0.031991 -0.046641
```

```
          0         1         2  ...         7         8         9
0  0.038076  0.050680  0.061696  ... -0.002592  0.019908 -0.017646
1 -0.001882 -0.044642 -0.051474  ... -0.039493 -0.068330 -0.092204
2  0.085299  0.050680  0.044451  ... -0.002592  0.002864 -0.025930
3 -0.089063 -0.044642 -0.011595  ...  0.034309  0.022692 -0.009362
4  0.005383 -0.044642 -0.036385  ... -0.002592 -0.031991 -0.046641

[5 rows x 10 columns]
```

If you don't see all columns, use the cursor to scroll to the right. Now let's check the number of columns and rows.

[62]:
```
no_rows = len(df_diabetes)
no_cols = len(df_diabetes.columns)

print('Rows:', no_rows, 'Columns:', no_cols)
```

```
Rows: 442 Columns: 10
```


```
Rows: 442 Columns: 10
```


There are 442 rows organised in 10 columns.

To get an overview, let us extract the mean of each column using 'describe' and plot all means as a bar chart. The Matplotlib function to plot a bar chart is `bar`:

[63]:
```
conc_means = list()

for col in df_diabetes:
    conc_means.append(df_diabetes[col].describe()['mean'])

print('The columns are: ', list(df_diabetes.columns))
print('The medians are: ', conc_means, 2)
```

```
The columns are:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
The medians are:  [-3.6342849293088766e-16, 1.3083425745511955e-16,
```

```
-8.045349203335693e-16, 1.2816545210746291e-16, -8.835315586242054e-17,
1.327024211984792e-16, -4.574646342983182e-16, 3.777301498233299e-16,
-3.8308542173050264e-16, -3.412882015081407e-16] 2

The columns are:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
The medians are:  [-3.6396225400041895e-16, 1.309912460049817e-16, -8.013951493363262e-16, 1.289
```

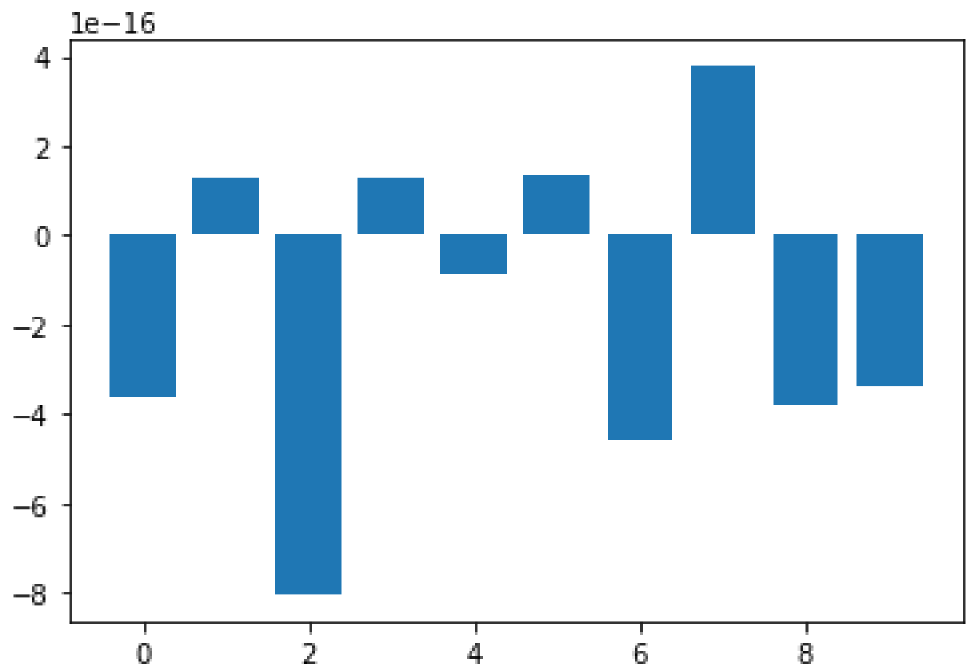
[64]:

```python
fig, ax = subplots()

bins = range(10)

ax.bar(bins, conc_means);

show()
```



The bars in this plot go up and down. Note, however, that the vertical axis has values ranging from $-10^{(-16)}$ to $+10^{(-16)}$. This means that for all practical purposes all means are zero. This is not a coincidence. The original values have been normalised to zero mean for the purpose of applying some machine learning algorithm to them.

In this example, we see how important it is to check the data before working with them.

9 Exercises

End of chapter Exercises Download the cervical cancer data set provided, import it using `read_csv`.

1. How many rows and columns are there?
2. How many columns contain floating point numbers (float64)?
3. How many of the subjects are smokers?
4. Calculate the percentage of smokers
5. Plot the age distribution (with e.g. 50 bins)
6. Get the mean and standard distribution of age of first sexual intercourse

10 Please check these solutions only after submitting the assignments.

10.1 Q1

[66]:
```
df_cervix = read_csv("data/cervical_cancer.csv")

df_cervix.head(10)


cervix_rows, cervix_cols = len(df_cervix), len(df_cervix.columns)

print('Number of rows:', cervix_rows)
print('Number of columns:', cervix_cols)
```

```
Number of rows: 668
Number of columns: 34

   Age  Number of sexual partners  ...  Citology  Biopsy
0   18                        4.0  ...         0       0
1   15                        1.0  ...         0       0
2   52                        5.0  ...         0       0
3   46                        3.0  ...         0       0
4   42                        3.0  ...         0       0
5   51                        3.0  ...         0       1
6   26                        1.0  ...         0       0
7   45                        1.0  ...         0       0
8   44                        3.0  ...         0       0
9   27                        1.0  ...         0       0

[10 rows x 34 columns]
Number of rows: 668
Number of columns: 34
```

10.2 Q2

[67]:
```
df_types = df_cervix.dtypes == 'float64'

print('There are', df_types.sum(), 'columns with floating point numbers')
```

```
There are 24 columns with floating point numbers
```


```
There are 24 columns with floating point numbers
```


[68]:
```
df_types
```

[68]:
```
Age                                   False
Number of sexual partners              True
First sexual intercourse               True
Num of pregnancies                     True
Smokes                                 True
Smokes (years)                         True
Smokes (packs/year)                    True
Hormonal Contraceptives                True
Hormonal Contraceptives (years)        True
IUD                                    True
IUD (years)                            True
STDs                                   True
STDs (number)                          True
STDs:condylomatosis                    True
STDs:cervical condylomatosis           True
STDs:vaginal condylomatosis            True
STDs:vulvo-perineal condylomatosis     True
STDs:syphilis                          True
STDs:pelvic inflammatory disease       True
STDs:genital herpes                    True
STDs:molluscum contagiosum             True
STDs:AIDS                              True
STDs:HIV                               True
STDs:Hepatitis B                       True
STDs:HPV                               True
STDs: Number of diagnosis             False
Dx:Cancer                             False
Dx:CIN                                False
Dx:HPV                                False
Dx                                    False
Hinselmann                            False
Schiller                              False
Citology                              False
Biopsy                                False
dtype: bool
```


```
Age                                   False
```

```
Number of sexual partners             True
First sexual intercourse              True
Num of pregnancies                    True
Smokes                                True
Smokes (years)                        True
Smokes (packs/year)                   True
Hormonal Contraceptives               True
Hormonal Contraceptives (years)       True
IUD                                   True
IUD (years)                           True
STDs                                  True
STDs (number)                         True
STDs:condylomatosis                   True
STDs:cervical condylomatosis          True
STDs:vaginal condylomatosis           True
STDs:vulvo-perineal condylomatosis    True
STDs:syphilis                         True
STDs:pelvic inflammatory disease      True
STDs:genital herpes                   True
STDs:molluscum contagiosum            True
STDs:AIDS                             True
STDs:HIV                              True
STDs:Hepatitis B                      True
STDs:HPV                              True
STDs: Number of diagnosis            False
Dx:Cancer                            False
Dx:CIN                               False
Dx:HPV                               False
Dx                                   False
Hinselmann                           False
Schiller                             False
Citology                             False
Biopsy                               False
dtype: bool
```

10.3 Q3

[69]:
```
for col in df_cervix:

    print(type(df_cervix[col][0]))


cervix_smoker = df_cervix['Smokes'] == 1.0
```

```
<class 'numpy.int64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
```

```
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>

<class 'numpy.int64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
```

```
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.float64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
<class 'numpy.int64'>
```

10.4 Q4

[70]:
```python
print('There are', cervix_smoker.sum(), 'smokers.')
print('This is', round(100*cervix_smoker.sum() / cervix_rows, 1), '% of the total.')
```

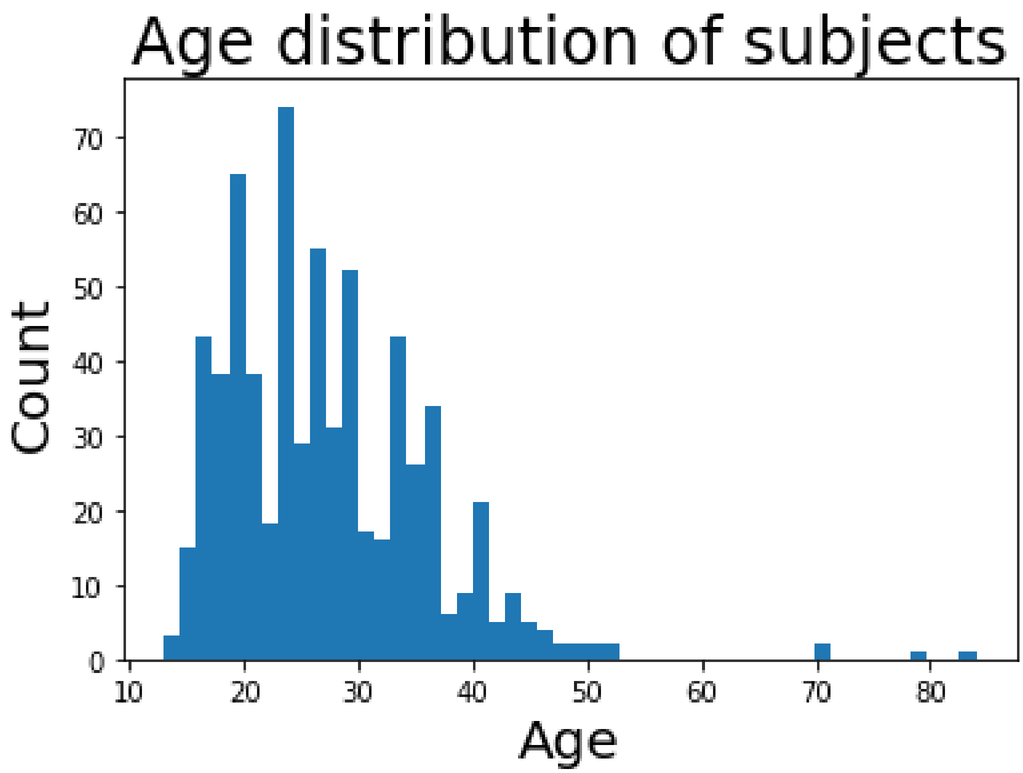
```
There are 96 smokers.
This is 14.4 % of the total.
```


```
There are 96 smokers.
This is 14.4 % of the total.
```


10.5 Q5

[71]:
```python
fig, ax = subplots()

ax.hist(df_cervix['Age'], bins=50)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylabel('Count', fontsize=20)
ax.set_title('Age distribution of subjects',  fontsize=24);
show()
```



10.6 Q6

[72]:
```
int_mean = df_cervix['First sexual intercourse'].mean()

int_std = df_cervix['First sexual intercourse'].std()

print('Mean of age of first sexual intercourse: ', round(int_mean, 1))
print('Standard distribution of age of first sexual intercourse: ',␣
 ↪round(int_std, 1))
```

```
Mean of age of first sexual intercourse:  17.1
Standard distribution of age of first sexual intercourse:  2.9
```


```
Mean of age of first sexual intercourse:  17.1
Standard distribution of age of first sexual intercourse:  2.9
```


- Pandas package contains useful functions to work with dataframes.
- `iloc` property is used to index and slice a dataframe.
- `describe` function is used to get a summary of basic data features.
- The simplest way of visualisation is to use Pandas functionality.
- `Matplotlib` is a comprehensive library for creating static, animated, and interactive visualizations in Python.