IMPORTING AND MANIPULATING TABULAR DATA

> by Dr Juan H Klopper

- Research Fellow
- School for Data Science and Computational Thinking
- Stellenbosch University

KLOPPER
RESEARCH GROUP

INTRODUCTION

Data Science by its name and nature requires us to have acces to data. We have learned that images, sounds files, text, and much more pieces of information can be represented as data. In this course, we concentrate on tabular data.

Tabular data is data in rows and columns, either extracted from an image, a database, or similar structures and represented in an array. An array is a set of values in rows and columns. As in the case of colour images, it can these rows and column can also be stacked *on top* of each other. We will consider only sincgle *stacks* with data in a spreadsheet. There is a fantastic package for importing such tabular data.

The **pandas** package has much to do with the success of Python as a programming language for Data Science. It is an enormous package and is used to import data, to manipulate data, to do calculations with data, and even create graphs and plots using the data.

In this notebook, we are going to get a glimpse into the usefulness of the pandas package by importing some data captured in a spreadsheet file. We will then extract some of the data that is of interest to us. In later notebooks, we will do all sorts of useful analysis on the extracted data.

PACKAGES FOR THIS NOTEBOOK

It is useful to import all packages at the start of a notebook. This allows us to keep track of what we are using in the notebook.

```
import pandas as pd  # Package to work with data
```

```
import numpy as np  # Numerical analysis package
```

To import a file from a Google Drive, we need a special function. This is not required when running Python on a local system, where we can simply refer to the *address* of the file on the internal (or network) drive.

```
from google.colab import drive  # Connect to Google Drive
```

Below, we us a magic command, `%load_ext` to load a `google.colab.data_table`. It produces better tables when using Colab and printing such tables to the screen.

```
# Format tables printed to the screen (don't put comment on the same line as the
%load_ext google.colab.data_table
```

IMPORTING DATA

In Google Colaboratory, we have to *mount* the cloud drive with our data file. As mentioned above, this setp is not required when using a local system. When running the cell below a link appears that you have to click on. A new tab will open up in your browser. You have to sign in to your Google account again giving permission to this Colab notebook to read files from your Google Drive (all in the name of security which is important). Once this is done a tab will open with a secuirty link that you have to copy (there is a convenient icon next to the secuirty code that will copy it). Copy it an close the tab. Then you have to paste the security key into the generated box below the code and hit enter or return.

```
drive.mount('/gdrive', force_remount=True)  # Connect to Google Drive
# With force_remount=True we can run this cell again later if needed
```

```
    Mounted at /gdrive
```

Now we navigate to the desired folder in Google Drive by specifying its address as a string. The address is a string and goes in a set of quotation marks. When you explore your own Google Drive simply note the location of a file if you have data stored somewher else

Note that you can also import a data file from your local system. There is a code snippet that you will find under the code snipper icon on the top left of this Colab notebook. The icon is the `<>` icon under the magnifying class for searches. You can scroll down the list to find other useful code snippets that you can use in your projects.

In the cell below, we see the `%cd` magic command that let's us change directory.

```
%cd '/gdrive/My Drive/Stellenbosch University/School for Data Science and Comput
```

```
/gdrive/My Drive/Stellenbosch University/School for Data Science and Computat
```

The `%ls` magic command will print a list of the files in the directory to which we changed into.

```
%ls
```

```
australia_rain.csv    crops.csv       DefaultMissingData.csv
bitcoin_ethereum.csv  customers.csv   kaggle_survey_2020_responses.csv
breast_cancer.csv     data.csv        MissingData.csv
client_data.csv       DatesTimes.csv  montague_gardens_construction.csv
```

We note that there is a csv file called `customer_data.csv`. We can import it using pandas' `read_csv()` function. Since it is not a Python function, we have to specify where (from what package) it came from. This is done by preceding the function with the pandas namespace abbreviation that we used initially, `pd`.

```
df = pd.read_csv('data.csv')  # Import the spreadsheet file
```

Since we navigated to this directory with the `%cd` magic command, we only need to type in the name and extension of your spreadsheet file (using quotation marks as it is a string).

The `type` function used below shows that the object assigned to the `df` computer variable is a DataFrame object.

```
type(df)  # Type of the object held in the computer variable df
```

```
pandas.core.frame.DataFrame
```

We can look at the attributes and methods of dataframe objects using Python's `dir` function.

```
dir(df)
```

```
 'rtruediv',
 'sBP',
 'sample',
 'select_dtypes',
 'sem',
 'set_axis',
 'set_index',
 'shape',
 'shift',
 'size',
 'skew',
 'slice_shift',
 'sort_index',
 'sort_values',
 'squeeze',
 'stack',
 'std',
 'style',
 'sub',
 'subtract',
 'sum',
 'swapaxes',
 'swaplevel',
 'tail',
 'take',
 'to_clipboard',
 'to_csv',
 'to_dict',
 'to_excel',
 'to_feather',
 'to_gbq',
 'to_hdf',
 'to_html',
 'to_json',
 'to_latex',
 'to_markdown',
 'to_numpy',
 'to_parquet',
 'to_period',
 'to_pickle',
 'to_records',
 'to_sql',
 'to_stata',
 'to_string',
 'to_timestamp',
 'to_xarray',
 'transform',

 'transpose',
 'truediv',
 'truncate',
 'tz_convert',
 'tz_localize',
 'unstack',
 'update',
 'value_counts',
 'values',
 'var',
 'where',
 'xs']
```

There is quite a lot of them. Thw first few are the statistical variables (column headers in the first row of the spreadsheet). We can explore each and every one of the rest of the methods and attributes on the Pandas page for [DataFrame objects](#).

Once such method is the `head` method. By default it returns the first five rows of a dataframe object. An integer value can be passed as argument if we need a different number of rows.

```
df.head()
```

1 to 5 of 5 entries Filter ?

index	Name	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	Ch
0	Dylan Patton	1981-10-07	43	Energy manager	0	47	145	1.2	1.2	1	
1	Sandra Howard	1993-01-27	53	Tax adviser	0	51	115	1.2	0.6	3	
2	Samantha Williams	1973-12-21	33	IT consultant	0	54	120	2.0	1.3	3	
3	Ashley Hensley	1981-12-01	43	Nurse, children's	0	54	103	2.1	1.6	4	

Since we used the `%load_ext google.colab.data_table` magic command at the start of the notebook, the dataframe object is printed to the screen in a very neat and useful way, allowing us to navigate the data.

The `shape` attribute (property) shows use the number of rows and columns, returned as a tuple. Note that unlike a method (which is like a Python function), an attribute has no parentheses.

```
df.shape   # Nuber of rows (subjects) and columns (statistical variables)
```

```
(200, 13)
```

There are 200 observations (rows) and 13 statistical variables (columns) in this *tidy* data set.

The `columns` property list all the column header names, called **labels**.

```
df.columns   # List the statistical variables
```

```
Index(['Name', 'DOB', 'Age', 'Vocation', 'Smoke', 'HR', 'sBP',
       'CholesterolBefore', 'TAG', 'Survey', 'CholesterolAfter', 'Delta',
       'Group'],
      dtype='object')
```

The majority of DataFrame objects will have two axes (rows and columns). We can verify this using the `ndim` attribute.

```
df.ndim
```

```
2
```

The `size` attribute gives us the total number of data point values (the product of the number of rows and columns).

```
df.size
```

```
2600
```

The last attribute that we will take a look at is the `dtype` attribute. It returns the Python data type of the values in each of the columns. This is a very important step. Pandas does its best to interpret the data type. Dependening on how the spreadsheet was created and how dat was entered, it is not always possible to correctly interpret the type. In this case we might have to change the data type. Remember that we base analysis of data on the dat type of the variable.

```
df.dtypes
```

```
Name                  object
DOB                   object
Age                    int64
Vocation              object
Smoke                  int64
HR                     int64
sBP                    int64
CholesterolBefore    float64
TAG                  float64
Survey                 int64
CholesterolAfter     float64
Delta                float64
Group                 object
dtype: object
```

Categorical variables are denoted as an `object` type. Numerical variable can be either integer or floating point numbers (numbers with decimal places). These are `int64` and `float64` (denoting 64-bit precision) respectively.

We refer to the data about data as meta data. It is important to view the meta data of any data that you import to make sure that the data did indeed import correctly and to start to learn a little bit about the data.

EXTRACTING ROWS AND COLUMNS

To analyse data, we want to extract only certain values. This is a very useful skill.

Pandas refers to a single column in a dataframe object as a **Series** object. We can also create standalone series objects, but in the context of analysing data, a standalone series object is perhaps not as useful. Below, we extract just the *Age* column (statistical variable) and save it as a series object. The notation uses square brackets, with the column name represented as a string.

```
age_column = df['Age'] # Note the use of square brackets
```

Our new object is indeed a series object.

```
type(age_column)
```

```
pandas.core.series.Series
```

Since we have no *illegal* characters in the column name such as spaces, we can also make use of dot notation. Below, we overwrite the `age_column` computer variable by reassigning it (using the same name).

```
age_column = df.Age # A shorter and more convenient way of extracting a column
```

We can display the first few rows in the series object with the `head` method.

```
age_column.head()
```

```
0    43
1    53
2    33
3    43
4    46
Name: Age, dtype: int64
```

Here we see further evidence that it is not just a Python list or a numpy array, but a series object, by noting the index column.

At times it may be more useful to work with a numpy array, rather than a pandas series. To extract the age values as a numpy array, we use the `.to_numpy` method.

```
age = df.Age.to_numpy()
```

The object assigned to the `age` computer variable is a numpy array.

```
type(age)
```

```
numpy.ndarray
```

As a numpy array, it has a number of attributes and methods. We use the `dir` function again to print out all the attributes and methods for this Python object type.

```
dir(age)
```

```
 'compress',
 'conj',
 'conjugate',
 'copy',
 'ctypes',
 'cumprod',
 'cumsum',
 'data',
 'diagonal',
 'dot',
 'dtype',
 'dump',
 'dumps',
 'fill',
 'flags',
 'flat',
 'flatten',
 'getfield',
 'imag',
 'item',
 'itemset',
 'itemsize',
 'max',
 'mean',
 'min',
 'nbytes',
 'ndim',
 'newbyteorder',
 'nonzero',
 'partition',
 'prod',
 'ptp',
 'put',
 'ravel',
 'real',
 'repeat',
 'reshape',
 'resize',
 'round',
 'searchsorted',
 'setfield',
 'setflags',
 'shape',
 'size',
 'sort',
 'squeeze',
 'std',
 'strides',
 'sum',
 'swapaxes',
 'take',
 'tobytes',
 'tofile',
 'tolist',
 'tostring',
 'trace',
 'transpose',
 'var',
 'view']
```

Below, we look at the minimum value and the maxiumum value in the `age` array, and calculate the average of all the values using the `.min`, the `.max`, and the `.mean` methods.

```
age.min() # The minimum value in the array
```

```
30
```

```
age.max() # The maximum value in the array
```

```
75
```

```
age.mean() # The mean of all the values in the array
```

```
53.07
```

We can specify inidividual rows (subjects) by making use of the `.iloc[]` attribute (or property, which is the term used by pandas) for a dataframe object. The `iloc` property stands for *integer location*, so we must use integers to specify the row and column numbers. We add an index value in square brackets for the property. Below, we extract the first row. Remember than Python is `0` indexed, so the first column has an index of 0.

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

```
Name                      Dylan Patton
DOB                         1981-10-07
Age                                 43
Vocation                Energy manager
Smoke                                0
HR                                  47
sBP                                145
CholesterolBefore                  1.2
TAG                                1.2
Survey                               1
CholesterolAfter                   0.7
Delta                              0.5
Group                           Active
Name: 0, dtype: object
```

We can specify certain rows by passing a list of integer values.

```
df.iloc[[2, 3, 5]] # Rows 3, 4, and 6 (remember, we are starting the indexing a
```

1 to 3 of 3 entries Filter ?

index	Name	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	Chol
2	Samantha Williams	1973-12-21	33	IT consultant	0	54	120	2.0	1.3	3	
3	Ashley Hensley	1981-12-01	43	Nurse, children's	0	54	103	2.1	1.6	4	
5	Leslie	1994-	48	Politician's	0	59	122	2.8	1.4	4	

Slicing is also allowed. This is done by specifying a range of values. The range object uses colon notation. Below, we use `0:2`. This includes the indices `0`, and `1`. The last index value in NOT included.

```
df.iloc[0:2]  # The first and second row
```

1 to 2 of 2 entries Filter ?

index	Name	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	Cholest
0	Dylan Patton	1981-10-07	43	Energy manager	0	47	145	1.2	1.2	1	
1	Sandra Howard	1993-01-27	53	Tax adviser	0	51	115	1.2	0.6	3	

The columns can also be indexed. Here we use the *row, column* notation. Below then, we extract the first five rows, but only for the *DOB* and *Age* variables, which are columns 1 and 2.

```
df.iloc[0:5,[1, 2]]
```

1 to 5 of 5 entries Filter ?

index	DOB	Age
0	1981-10-07	43
1	1993-01-27	53
2	1973-12-21	33
3	1981-12-01	43
4	1964-06-23	46

Show 25 per page

Above, we passed the rows as a range and the two columns as a list.

The `.loc[]` property can be used in a similar fashion. Here we can specify the column names (as a list or a slice). If the index values were not integers, but strings, we could also use those names. Remeber that the row and column names are referred to as **labels**. Below, we extract the same labels as we did above. Note, though, that the range includes the sixth row. When extracting rows and column, ALWAYS use the `row, column` notation. Since we want two columns, we pass them as a Python list object (in square brackets) after the comman. Each column name is passed as a string.

```
df.loc[0:5, ['DOB', 'Age']]
```

1 to 6 of 6 entries Filter

index	DOB	Age
0	1981-10-07	43
1	1993-01-27	53
2	1973-12-21	33
3	1981-12-01	43
4	1964-06-23	46
5	1994-08-25	48

Show 25 per page

The `.iat` indexing extracts a single *cell* by using its row and column index.

```
df.iat[3, 2]
```

```
43
```

There is also an `at[]` indexing, which does the same. Here we can specify labels, though.

FILTERING DATA

Filtering data is one of the most useful things that we can do with data in a dataframe object. In this section, we will start to learn how to filter data by extracting numpy array objects based on criteria that we which to investigate or by creating brand new dataframes.

In order to do filtering we use conditionals. We have learned about these in the prviosu notebook. For instance, below we ask if 3 is greater than 4 and then if 3 is equal to 3.0.

```
# A conditional returns a True or False value
3 > 4
```

```
False
```

```
# The double equal symbols conditional
3 == 3.0
```

```
True
```

FINDING UNIQUE VALUES IN A COLUMN

Remember that we refer to the **sample space** of a variable as all the possible values that a variable can take. This is particulary useful when looking at categorical variables. The `unique` method is used to find all the sample space elements in a column

```
df.Smoke.unique() # Data entries encoded as 0, 1, and 2

    array([0, 2, 1])
```

We note that there are three elements in the sample space of this column. This method is great for *surprises* that might be hidden in a dataframe such as one or more strings in a numerical data column. A common example would be the *Age* column that has one or two strings such as *thirty-two* in it, instead of 32. Strings in a numerical data column will prevent calculations on that column and such errors in the data must be corrected. We will learn how to change values using the `replace` method later in this notebook.

AGES OF ALL NON-SMOKERS

The `Smoke` column contain information about the smoking habits of the respondents in the data set. We have seen above that the sample space contains three integers, `0` for not smoking, `1` for smoking, and `2` for previous smoking.

Here, we are interested in creating an array that contains the ages of only the patients who do not smoke in our dataframe. To do this, we use indexing directly. A conditional is used to include only *0* patients (`df.Smoke == 0`). We then reference the column that we are interested in, which is `Age`, followed by the `to_numpy` method.

```
non_smoker_age = df[df.Smoke == 0]['Age'].to_numpy()
non_smoker_age # Print the values to the screen

    array([43, 53, 33, 43, 46, 48, 54, 58, 44, 31, 45, 35, 49, 56, 57, 35, 50,
           49, 63, 45, 51, 40, 47, 41, 47, 38, 54, 30, 46, 64, 40, 45, 65, 55,
           53, 54, 72, 32, 38, 59, 53, 42, 38, 51, 37, 36, 48, 49, 62, 39, 74,
           42, 72, 61, 33, 30, 44, 71, 49, 75, 43, 55, 38, 36, 46, 60, 57, 69,
           56, 66, 60, 42, 32, 31, 56, 35, 63, 54, 68, 72, 40, 54, 62, 74, 62,
           41, 61, 61])
```

When first using this code, it may seem a bit difficult. It does read rather like an English language sentence, though. *Take the dataframe object. Extract the rows in column `Smoke` that are `0`. For all of these rows return the `Age` values as a numpy array*.

As an alternative, we can use the `loc` indexing, passing a *row* and a *column* specification as arguemnts. The *row* interrogates the `Smoke` column and includes only those with a `0` entry. The *column* is then specified to the the `Age` column.

```
df.loc[df.Smoke == 0, 'Age'].to_numpy()
```

```
array([43, 53, 33, 43, 46, 48, 54, 58, 44, 31, 45, 35, 49, 56, 57, 35, 50,
       49, 63, 45, 51, 40, 47, 41, 47, 38, 54, 30, 46, 64, 40, 45, 65, 55,
       53, 54, 72, 32, 38, 59, 53, 42, 38, 51, 37, 36, 48, 49, 62, 39, 74,
       42, 72, 61, 33, 30, 44, 71, 49, 75, 43, 55, 38, 36, 46, 60, 57, 69,
       56, 66, 60, 42, 32, 31, 56, 35, 63, 54, 68, 72, 40, 54, 62, 74, 62,
       41, 61, 61])
```

The different ways to interact with pandas adds to its power and you can find a way to achieve your data analysis goals that best first your way of work.

Since this is now a numpy array object, we can use methods such as the `mean` method to calculate the average age of all the non-smoking participants.

```
non_smoker_age.mean()
```

```
50.09090909090909
```

NON-SMOKER AGES WHERE SURVEY CHOICE IS 3

We now need to filter by two criteria (two columns), `Age` and `Survey`. The filtering can either refer to **and** or **or**. In the first, we require all the criteria to be met and in the second, only one of the criteria need be met (return a `True` value).

The symbol for **and** is `&` and for **or** is `|`. Below, we use `&` since we want both criteria to be met. Each filter is created in a set of parentheses. the code uses the `row, column` notation.

```
non_smoker_satisfied_age = df.loc[(df.Smoke == 0) & (df.Survey == 3), 'Age'].to_
```

In English the code reads: *Take the `df` dataframe object and look down the rows of the `Smoke` and `Survey` columns. Return only the rows where `Smoke` is `0` AND `Survey` is `3`. Then return the `Age` column for all these rows fulfilling both criteria*.

NEVER SMOKED OR SATISFACTION SCORE GREATER THAN 3

We are interested in those participants who never smoked OR those that have a satisfaction score of more than `3`. Here our filtering criteria requires only *one* of the two criteria to return `True`. A clearer way to build these filtering criteria, is to save them as a computer variable first.

```
# Saving the filtering criteria as a computer variable
# The > symbol is used to indicate greater that 3
crit = (df.Smoke == 0) | (df.Survey > 3)
```

We can now pass this to the `loc` property (as row and then specify the column name).

```
non_smoker_or_satisifed_age = df.loc[crit, 'Age'].to_numpy()
```

NON-SMOKERS AND SATISFACTION SCORE OF 3 OR LESS

Non-smokers are either those who have never smoked (`0`) or those that are ex-smokers (`2`). Both need inclusion. In other words, we need to exclude the smokers. It is easier to select just one element. One way to deal with the interrogation of the data is through negation. We can change our language into an opposte view, i.e start with filtering the current rows with a score of greater than 3. Then we simply use negation with the tilde, `~`, symbol to exclude these cases.

```
# Include those who do not smoke and have a score of more than 3
crit = (df.Smoke == 1) & (df.Survey > 3)

# Now we exclude these rows with a ~ negation symbol
not_no_smoker_satisfied_age = df.loc[~crit, 'Age'].to_numpy()
not_no_smoker_satisfied_age
```

```
array([43, 53, 33, 43, 46, 48, 54, 58, 44, 31, 45, 35, 49, 56, 57, 38, 35,
       50, 45, 49, 63, 45, 51, 43, 31, 58, 40, 47, 45, 41, 47, 38, 54, 30,
       46, 64, 40, 45, 65, 74, 55, 58, 53, 68, 54, 72, 32, 38, 59, 53, 42,
       67, 38, 51, 37, 36, 34, 31, 48, 49, 62, 39, 74, 42, 60, 67, 42, 52,
       37, 61, 72, 49, 49, 49, 43, 38, 45, 71, 73, 60, 70, 49, 65, 30, 50,
       69, 72, 61, 33, 30, 44, 71, 49, 75, 43, 35, 55, 38, 36, 46, 60, 40,
       57, 69, 56, 66, 60, 42, 49, 68, 32, 31, 56, 35, 63, 54, 68, 72, 40,
       41, 66, 73, 36, 57, 74, 48, 72, 71, 42, 54, 58, 60, 63, 44, 55, 43,
       71, 54, 61, 55, 67, 68, 71, 54, 57, 45, 62, 33, 74, 43, 35, 64, 55,
       74, 50, 40, 62, 63, 68, 41, 72, 61, 70, 61, 65, 66])
```

CREATE A NEW DATAFRAME OBJECT THAT ONLY CONTAINS PARTICIPANTS YOUNGER THAN 50

Instead of just an array of values, we want to create a new dataframe object. (Because it is a part of an existing dataframe object, some Data Scientist refer to it as a sub-dataframe object.) It includes all the columns (variables), but only for patients up to and including 49 years of age. This is very simple to achieve.

```
new_df = df[df.Age < 50]
new_df.head()
```

1 to 5 of 5 entries Filter ?

index	Name	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	Ch
0	Dylan Patton	1981-10-07	43	Energy manager	0	47	145	1.2	1.2	1	
2	Samantha Williams	1973-12-21	33	IT consultant	0	54	120	2.0	1.3	3	

Let's verify the result of our code by looking at the maximum *Age* value of this new dataframe. Below, we see three ways to return the maximum value in the new *Age* column.

```
new_df.Age.max()  # Using the column name directly
```

```
49
```

```
new_df['Age'].max()  # Using the column name as a column index name
```

```
49
```

```
new_df.loc[:, 'Age'].max()  # Using the loc property
```

```
49
```

Above we see the shorthand notation for including *all* elements, the colon, `:`. Since this is the `.loc[]` property, we expect row and column labels. For the rows then, we use the colon symbol to indicate that we are interested in all the rows. After the comma we indicate the column label and outside of the `.loc[]` indexing, we use the `.max()` method.

CREATE A NEW DATAFRAME FOR PARTICIPANTS WITH A RESTRICTED LIST OF JOB TITLES

Up until now, we have had single values for our filter criteria, even though we had multiple criteria. Here we might want to filter on more than one value, say *IT consultant*, *Energy manager*, and *Clinical embryologist*. Since the sample space is quite large, negation would not be a good solution as we would need to list all the other `Vocation` sample space values. Here's how we would create the new dataframe object, by making use of the `isin` method.

We create a list of the sample space elements that we are interested in. We then build a criterium using the `isin` method. Its job is exactely what it sounds like, *is in*, i.e. select only an element that *is in* the list.

```
# Create a Python list object with all the column names
jobs = ['IT consultant', 'Energy manager', 'Clinical embryologist']

# Build a criterium
```

```
crit = df.Vocation.isin(jobs)

# Create the new dataframe object and print the first 5 rows to the screen
jobs_df = df.loc[crit]
jobs_df.head()
```

1 to 4 of 4 entries Filter ?

index	Name	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	Ch
0	Dylan Patton	1981-10-07	43	Energy manager	0	47	145	1.2	1.2	1	
2	Samantha Williams	1973-12-21	33	IT consultant	0	54	120	2.0	1.3	3	
4	Robert Wilson	1964-06-23	46	Clinical embryologist	0	61	138	2.8	2.1	5	
188	Joan Chavez	1999-10-07	41	Energy manager	0	93	182	9.1	5.0	2	

Show 25 per page

CREATE A NEW DATAFRAME WHERE THE WORD *MANAGER* APPEARS IN THE `VOCATION` COLUMN

This filter uses a string method, `str.contains`. It is ideal for free-form input cells in a spreadsheet, where we can search for keywords. Below, we see an extra `na=False` argument. This is used to deal with dataframe obejcts with missing data. We will learn how to deal with missing data later.

```
# Build a criterium with the str.contains method
crit = df.Vocation.str.contains('manager', na=False)

# Create the new dataframe object and print the first 5 rows to the screen
vocation_df = df.loc[crit]
vocation_df.head()
```

1 to 5 of 5 entries Filter ?

index	Name	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	Cl
0	Dylan Patton	1981-10-07	43	Energy manager	0	47	145	1.2	1.2	1	
34	Mr. Tyler Strickland DDS	1940-08-27	46	Tourist information centre manager	0	62	136	4.1	2.3	2	
54	Stephanie Jacobs	1977-06-18	38	Estate manager/land agent	0	69	139	4.4	2.7	2	
55	Juan Johnson	1956-12-09	51	Logistics and distribution manager	0	65	141	4.5	2.9	5	

We note that the term *manager* appear in all the values for the *Vocation* column.

UPDATING OR CHANGING THE VALUES IN A DATAFRAME

Another valueble skill is to be able to change actual data in a dataframe object. Fortunately, datadrame objects can be manipulated in many ways. We begin by looking at changes to the column names.

RENAMING COLUMNS

We can replace the names of individual columns with the `rename` method using a dictionary. Below we change *Name* to *Participant*. For changes to be permanent we need to change the default `inplace` argument value to `True`.

```
df.rename(columns={'Name':'Participant'}, inplace=True)
df.columns
```

```
Index(['Participant', 'DOB', 'Age', 'Vocation', 'Smoke', 'HR', 'sBP',
       'CholesterolBefore', 'TAG', 'Survey', 'CholesterolAfter', 'Delta',
       'Group'],
      dtype='object')
```

```
df.head()
```

1 to 5 of 5 entries Filter

index	Participant	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	C
0	Dylan Patton	1981-10-07	43	Energy manager	0	47	145	1.2	1.2	1	
1	Sandra Howard	1993-01-27	53	Tax adviser	0	51	115	1.2	0.6	3	
2	Samantha Williams	1973-12-21	33	IT consultant	0	54	120	2.0	1.3	3	
3	Ashley Hensley	1981-12-01	43	Nurse, children's	0	54	103	2.1	1.6	4	

ADD 2 TO EACH AGE VALUE

In specific types of research, personal data are obfuscated to protect the privacy of the people in the dataset. In a simple case, we might decide to subtract 2 from the age of every patient. In reality, they are all 2 years older. To fix this prior to data analysis, we must add 2 to each age.

There are more than one way to achieve our goal. One way is to create a function and then use the `apply` method to apply the function to the values in a column.

User-defined functions are created using thd `def` and `return` keywords. The former tells Python that we are about the create a new function. After a space follows the name we want to givw to our function. A set of parentheses follow that contains a placeholder for the argument. In its simplest form, the latter keyword follows. As the name indicates thsi section returns a value for our function. Below, it is clear from the code that `x` will hold an argument value. The function then adds 2 to the argument value.

```
def add2(x):
  return x + 2
```

The first five age values are printed below using the `head` method for the `df.Age` series.

```
df.Age.head()  # Before
```

```
0    43
1    53
2    33
3    43
4    46
Name: Age, dtype: int64
```

The `apply` method is now used and the `add2` function is used. The value in each row is now increased by 2.

```
df.Age = df.Age.apply(add2)
df.Age.head()  # After
```

```
0    45
1    55
2    35
3    45
4    48
Name: Age, dtype: int64
```

The `lambda` function in Python is a quick albeit more advanced way to achieve our goal. It create a nameless function. Below, we subtract 2 from every *Age* entry to get back to where we started.

```
df.Age = df.Age.apply(lambda x: x - 2)
df.Age.head()
```

```
0    43
1    53
2    33
3    43
4    46
Name: Age, dtype: int64
```

The simplest way to add a value would be to simply refer to a column (a series object) and overwrite it. Remember that the `=` assignment operator assigns what is to its right to what is to its left. Below, we use series notation to overwrite the `Age` column by adding 2 to each row.

```
df.Age = df.Age + 2
```

CHANGE NOMINAL VARIABLE TO ORDINAL VARIABLE

For the purposes of encoding, we might want to change all *Active* values to 0 and *Control* values to 1 in the *Group* column. To do this, we could use the `map` method and then pass a dictionary object as argument. The dictionary holds key value pairs. The key is the old value and the value is the new value.

```
df.Group = df.Group.map({'Control':0, 'Active':1})
df.Group.head()
```

```
0    1
1    1
2    1
3    1
4    1
Name: Group, dtype: int64
```

One *problem* with the `map` method is that it will delete the entries (rows) for values that we do not specify. To keep the original data when not specified, we can use the `replace` method instead.

CHANGING COLUMNS

Adding columns is an often used technique when changing column. It is as simple as stating the new name in square brackets as a string and then adding a list of values. We need it to be the same length (number of rows) as the dataframe.

SPLITTING THE `PATIENT` COLUMN INTO A `FirstName` and `LastName` COLUMN

Below, we create two new columns called *FirstName* and *LastName* from the *Participant* column, splitting on the space using the `str.split` method.

```
new_data = df.Participant.str.split(' ', expand=True)
```

```
df['FirstName'] = new_data[0]
df['LastName'] = new_data[1]
df.head()
```

1 to 5 of 5 entries Filter ?

index	Participant	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey	C
0	Dylan Patton	1981-10-07	45	Energy manager	0	47	145	1.2	1.2	1	
1	Sandra Howard	1993-01-27	55	Tax adviser	0	51	115	1.2	0.6	3	
2	Samantha Williams	1973-12-21	35	IT consultant	0	54	120	2.0	1.3	3	
3	Ashley Hensley	1981-12-01	45	Nurse, children's	0	54	103	2.1	1.6	4	
4	Robert Wilson	1964-06-23	48	Clinical embryologist	0	61	138	2.8	2.1	5	

Show 25 per page

We can also combine two columns into one. Below, we use string concatination, combining the last name, a comma with a space (as a string) and the first name.

```
df['Name'] = df.LastName + ', ' + df.FirstName
df.Name.head()
```

```
0          Patton, Dylan
1         Howard, Sandra
2     Williams, Samantha
3        Hensley, Ashley
4         Wilson, Robert
Name: Name, dtype: object
```

CREATE A CATEGORICAL VARIABLE FROM NUMERICAL DATA

Below, we create three sample space elements: *low*, *intermediate*, and *high* for the *CholesterolBefore* value of each patient. To do so, we use the pandas `cut` function with specified bins. To understand the concept of bins, we start by looking at the minimum and maximum values.

```
df.CholesterolBefore.min()
```

```
1.2
```

```
df.CholesterolBefore.max()
```

```
11.1
```

With the `bins=3` argument, we create three equally sized bins in the range form 1.2 to 11.1.

In the code below, we create a new variable called `CholesterolBeforeLevel`. We use the pandas `cut` function with three arguments. The first is a pandas series object (the colum of interest). The second is the number of bins and the last is a list of names for each of the three bins.

```
# Non-physiological binning of cholesterol values
df['CholesterolBeforeLevel'] = pd.cut(df.CholesterolBefore, bins=3, labels=['low
```

Below, we view the first 10 new categorical values and actual values as a numpy array object.

```
df[['CholesterolBefore', 'CholesterolBeforeLevel']].head(10).to_numpy()
```

```
array([[1.2, 'low'],
       [1.2, 'low'],
       [2.0, 'low'],
       [2.1, 'low'],
       [2.8, 'low'],
       [2.8, 'low'],
       [2.9, 'low'],
       [3.1, 'low'],
       [3.1, 'low'],
       [3.2, 'low']], dtype=object)
```

These three bins are non-physiological in that we have specific values for low, normal, and high levels of cholesterol. To control the bin values, we can specify the bin cut-off values as a list. To understand this we need to know about open and closed intervals. An open interval such as $(10, 20)$ means that neither 10 nor 20 are included. Inclusion requires a closed interval, denoted by $[10, 20]$. We also have half-open intervals. In $(10, 20]$, 10 is not included but 20 is. We can also have $(10, 20]$, with 10 not included and 20 being included.

Consider then the values $11.2, 12.2, 13.2, 15, 16, 16, 19.2, 20$. Imagine then that everything below 13 is *low*, 13 to below 16 is normal and 16 and above is high. Proper intervals would then be $[11.2, 13)$. Here we use the lowest value as the inlcuded lower bound, but 13 is not included. A value of 13 would be in the second bin. The second bin would have bounds $[13, 16)$ and the last $[16, 20]$.

In pandas we can only have left or right half-open intervals. The keyword argument `right` is set to `False` by default, with right open intervals.

There is a `right=True` argument value. It states that the intervals are right-closed so that `bins=[10,20,30]` would mean intervals (10,20] 10 not being included but 20 being included here and (20,30] 20. Similarly the `include_lowest=False` argument means that the left-most value is not included (the 10 in this explanation). Set the argument to `True` to have the first

Below, we create three bins with intervals low = $[0, 2.5)$, normal = $[2.5, 5.0)$, and high = $[5.0, 20)$. So, if a patient has a cholesterol value of 5, they would fall in the high group. Note that 20 is above the maximum value and is a safe value to use. Note also that there are four numbers for three bins.

```
df.CholesterolBeforeLevel = pd.cut(df.CholesterolBefore,
                                   bins=[0,5,10,20],
                                   right=False,
                                   labels=['low', 'normal', 'high'])
```

DELETE A COLUMN

Deleting a column can be achieved using the `drop` method. To make the deletion permamant, we use the `inplace=True` argument. Let's delete our newly created *Name* column.

```
df.drop(columns=['Name'], inplace=True)
```

```
df.columns
```

```
Index(['Participant', 'DOB', 'Age', 'Vocation', 'Smoke', 'HR', 'sBP',
       'CholesterolBefore', 'TAG', 'Survey', 'CholesterolAfter', 'Delta',
       'Group', 'FirstName', 'LastName', 'CholesterolBeforeLevel'],
      dtype='object')
```

SORTING

Sorting can be a useful way to interact with our data. Below, we change the dataframe object by sorting the *LastNames* alphabetically. All the coresponing column will change as well, so that each row still pertains to the same patient.

```
df.sort_values(by='LastName')
```

1 to 25 of 200 entries Filter ?

index	Participant	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey
53	Christopher Abbott	1963-06-12	69	Ergonomist	2	66	128	4.4	2.6	2
170	Mary Aguilar	1952-01-09	64	Furniture designer	0	88	182	8.3	5.1	2
22	James Aguilar	2000-11-05	53	Immunologist	0	64	134	3.7	2.0	1
81	Marissa Anderson PhD	1940-05-15	51	Nurse, learning disability	1	75	170	5.0	2.7	3
152	Andrea Anderson	1974-04-08	75	Horticultural consultant	1	85	183	7.9	4.2	4
160	Elizabeth Ashley	1949-11-23	69	Newspaper journalist	1	87	187	8.1	4.3	2
112	Gregory Avila	1956-09-30	38	Trade mark attorney	0	78	168	6.3	3.9	3
43	Joshua Avila	1978-01-19	55	Media planner	0	64	129	4.3	2.5	2
23	Michael Banks	1977-08-19	45	Exhibition designer	2	62	132	3.7	2.6	3
171	Mary Barnett	1993-10-21	35	Operational investment banker	1	88	179	8.3	4.9	3
199	Julie Barrett	1972-07-27	68	Theme park manager	1	102	208	11.1	5.7	2
178	Jonathan Bautista	1938-06-28	66	Landscape architect	1	89	198	8.5	4.8	1
82	Justin Bennett	1963-04-04	71	Insurance broker	1	66	142	5.0	3.0	4
142	Nichole Best	1954-10-31	56	Careers information officer	1	82	175	7.6	3.9	5
127	Michael Black	1973-03-18	37	Fish farm manager	0	85	178	7.1	3.9	4
183	Leah Blankenship	1938-09-22	64	Dramatherapist	0	90	176	8.7	5.3	3
44	Kenneth Bowman	1934-07-13	70	Magazine features editor	1	68	135	4.3	3.1	1
33	Kyle Boyd	1959-12-30	32	Waste management officer	0	63	133	4.0	2.5	5
168	Angela Boyer	1977-06-11	59	Dancer	1	88	174	8.2	5.0	2
58	John Boyle	1979-07-30	36	Pharmacist, community	1	66	148	4.5	3.2	2
20	Mr. Bradley	1990-	65	Conservation	0	64	136	3.7	2.0	4

The alphabetical order can be reversed by using the `acending=False` argument.

```
df.sort_values(by='LastName', ascending=False)
```

1 to 25 of 200 entries Filter ?

index	Participant	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG
100	Kristina Zimmerman	1994-01-01	74	Agricultural consultant	0	70	157	5.7	3.6
6	Frank Zimmerman	1981-03-04	56	Police officer	0	60	129	2.9	2.4
60	Janet Young	1981-03-18	33	Aeronautical engineer	1	65	133	4.5	2.3
185	James Wright	1997-09-20	65	Advertising account executive	1	87	193	8.8	5.3
157	Jodi Wood	1946-11-29	56	Animal technologist	0	87	173	8.0	4.0
184	Angela Wilson	1988-05-19	67	Building control surveyor	1	90	190	8.8	4.5
191	Angela Wilson	1983-08-24	67	Designer, television/film set	1	92	202	9.3	5.0
179	Steven Wilson	1947-02-17	57	Surveyor, planning and development	1	89	194	8.6	5.3
4	Robert Wilson	1964-06-23	48	Clinical embryologist	0	61	138	2.8	2.1
97	Jason Williams	1944-12-22	60	Herpetologist	1	71	148	5.6	3.4
24	Clifford Williams	1957-05-04	33	Special effects artist	2	60	134	3.7	2.5
2	Samantha Williams	1973-12-21	35	IT consultant	0	54	120	2.0	1.3
159	Bob Williams	1991-03-10	57	Commercial/residential surveyor	1	82	167	8.1	4.1
120	Jonathan Williams	1982-08-12	62	Museum/gallery conservator	0	78	160	6.8	4.4
21	Lacey Wilcox	1998-02-24	47	Chemist, analytical	0	60	115	3.7	2.1
47	Michael White	1954-11-24	34	Museum/gallery exhibitions officer	0	63	137	4.3	3.0
73	Daniel White	1975-03-23	39	Brewing technologist	1	65	136	4.9	2.5
180	Paula White	1979-03-04	76	Engineer, control and instrumentation	0	89	187	8.6	4.6
10	James Wells	1998-08-09	47	Charity fundraiser	0	62	121	3.2	1.7
86	Jeffrey Washington	1994-12-04	47	Waste management officer	1	67	154	5.1	2.8
93	Shari Wagner	2001-08-29	51	Paramedic	2	68	149	5.3	3.6
143	Jeremy Wagner	1938-12-10	73	Cytogeneticist	1	104	205	7.7	4.3
175	Nicole Vance	1990-07-17	45	Personnel officer	1	87	193	8.4	5.0
30	Brittany Valenzuela	1992-08-02	49	Therapist, horticultural	0	59	132	3.9	2.6
34	Mr. Tyler Strickland DDS	1940-08-27	48	Tourist information centre manager	0	62	136	4.1	2.3

We can sort by more than one column at a time. This is done by passing a list of column names. Below, we sort by *Age* and the *sBP*. With default values, numerical and date values will be from smaller to larger values and from earlier to later dates and categorical variables will be alphabetical

```
df.sort_values(by=['Age', 'sBP'])
```

1 to 25 of 200 entries Filter ?

index	Participant	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey
33	Kyle Boyd	1959-12-30	32	Waste management officer	0	63	133	4.0	2.5	5
96	Brandi Ibarra	1973-11-01	32	Communications engineer	1	72	159	5.5	3.7	7
103	Mary Rodriguez	2001-07-07	32	Music tutor	0	74	168	5.9	3.4	4
9	Andrea Fletcher	1955-12-23	33	Lexicographer	0	59	122	3.2	1.7	5

The three participants aged 30 now have their systolic blood pressure values in ascending order.

24	Clifford		33	Special effects	2	60	134	3.7	2.5	4

Not all the column names passed as a list to sort by, need be in the same order. We can also pass a list with corresponding order.

47	Michael	1954-	34	exhibitions	0	63	137	4.3	3.0	5

```
# Sort Age in ascending and sBP in descending order
df.sort_values(by=['Age', 'sBP'], ascending=[True, False])
```

1 to 25 of 200 entries Filter ?

index	Participant	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey
103	Mary Rodriguez	2001-07-07	32	Music tutor	0	74	168	5.9	3.4	4
96	Brandi Ibarra	1973-11-01	32	Communications engineer	1	72	159	5.5	3.7	1
33	Kyle Boyd	1959-12-30	32	Waste management officer	0	63	133	4.0	2.5	5
125	Victoria Gordon	1956-08-05	33	Pharmacist, community	0	83	165	7.0	4.2	2
24	Clifford Williams	1957-05-04	33	Special effects artist	2	60	134	3.7	2.5	4
60	Janet Young	1981-03-18	33	Aeronautical engineer	1	65	133	4.5	2.3	3
9	Andrea Fletcher	1955-12-23	33	Lexicographer	0	59	122	3.2	1.7	5
124	Renee Schneider	1948-09-24	34	Corporate treasurer	0	83	170	7.0	4.1	2
47	Michael White	1954-11-24	34	Museum/gallery exhibitions officer	0	63	137	4.3	3.0	5
171	Mary Barnett	1993-10-21	35	Operational investment banker	1	88	179	8.3	4.9	3

The three patients aged 30 are now sorted by the highest systolic blood pressure first.

The `sort_value` method does not make permanent changes to the dataframe, unless the argument `inplace` (which is set to `False` by default) is set to `True`.

The `nlargest` method is useful if we only want to view the highest numerical values in a column. Below, we look at the 15 highest systolic blood pressure values.

```
1 df.sBP.nlargest(15)
```

```
197    212
199    208
143    205
190    203
194    203
191    202
195    201
198    200
174    198
178    198
189    198
166    196
173    195
163    194
179    194
Name: sBP, dtype: int64
```

We can reverse the order of the syntax above a bit, if we want to see the rest of the columns too.

```
# Column is listed as arguemnt
df.nlargest(10, 'sBP')
```

1 to 10 of 10 entries Filter ?

index	Participant	DOB	Age	Vocation	Smoke	HR	sBP	CholesterolBefore	TAG	Survey
197	Charles Smith	1959-01-30	63	Chartered certified accountant	0	99	212	10.1	5.6	4
199	Julie Barrett	1972-07-27	68	Theme park manager	1	102	208	11.1	5.7	2
143	Jeremy Wagner	1938-12-10	73	Cytogeneticist	1	104	205	7.7	4.3	3
190	Rachel Mcguire	1970-12-23	64	Medical sales representative	1	92	203	9.3	5.1	4
194	Jeffery Silva	1973-11-25	72	Bookseller	1	94	203	9.9	5.4	1
191	Angela Wilson	1983-08-24	67	Designer, television/film set	1	92	202	9.3	5.0	5
195	John Curtis	1936-11-25	68	Sales professional, IT	1	96	201	10.1	5.1	5
198	Barry Porter	1979-05-30	67	Dancer	1	98	200	10.1	5.3	3
174	Heidi Gaines	1974-06-26	66	Occupational therapist	1	89	198	8.4	4.5	4
178	Jonathan Bautista	1938-06-28	66	Landscape architect	1	89	198	8.5	4.8	1

Show 25 per page

If we want the smallest values, there is also a `nsmallest` method.

MISSING VALUES

THE NUMPY `nan` VALUE

It is very often that datasets contain missing data. The numpy library has a specific entity called a `nan` value. This stands for *not a number*. Below, we see it by itself and also as an element in a Python list.

```
np.nan
```

```
nan
```

```
my_list = [1, 2, 3, np.nan]
my_list
```

```
[1, 2, 3, nan]
```

The list object, `my_list`, above, cannot be used as argument to functions such as `sum`, since Python does not know how to deal with this missing data. Below, we use the numpy `sum` function. The results is a `nan` value.

```
np.sum(my_list)
```

```
nan
```

Now, let's have a look at how pandas deals with misssing values. We will import another spreadsheet file that contains missing data.

A DATAFRAME WITH MISSING DATA

```
missing_df = pd.read_csv('MissingData.csv')
```

The DataFrame has the following columns: `age`, `salary`, and `previous_company` Most of the columns are self-explanatory. The *previous_company* indicates whether the person had previously used a different investment company instead of ours or had no investment at all.

When we print the dataframe object, we note all the `NaN` values, which pandas uses to indicate missing data.

```
missing_df
```

1 to 25 of 32 entries Filter

index	age	salary	previous_company
0	57.0	NaN	1.0
1	56.0	50927.0	NaN
2	46.0	75500.0	3.0
3	NaN	84417.0	NaN
4	60.0	63002.0	1.0
5	54.0	54652.0	NaN
6	NaN	65739.0	1.0
7	64.0	89397.0	3.0
8	60.0	77797.0	4.0
9	61.0	NaN	1.0
10	51.0	92767.0	5.0
11	NaN	59873.0	2.0

DELETING MISSING DATA

The first way of dealing with missing data, is to simply remove all the rows that contain any missing data. This is done with the `.dropna()` method. To make the changes permanent, we would have to use the `inplace=True` argument. Instead of permanent removal, we create a new dataframe object.

```
complete_data_df = missing_df.dropna() # Non permanent removal
complete_data_df
```

1 to 19 of 19 entries Filter

index	age	salary	previous_company
2	46.0	75500.0	3.0
4	60.0	63002.0	1.0
7	64.0	89397.0	3.0
8	60.0	77797.0	4.0
10	51.0	92767.0	5.0
12	49.0	91001.0	4.0
13	59.0	54212.0	4.0
14	61.0	84423.0	2.0
17	48.0	63824.0	5.0
18	53.0	53143.0	2.0
20	55.0	79309.0	2.0
22	49.0	52895.0	1.0
23	57.0	54035.0	3.0
24	58.0	64962.0	3.0
26	63.0	63861.0	2.0
27	47.0	55188.0	3.0
28	63.0	61220.0	3.0
30	49.0	63398.0	1.0
31	57.0	94900.0	5.0

Show 25 per page

There is another argument for this method, `how` that is set to `any`. This default states that if any of the values in a row are missing, the whole row is dropped. There is also an `all` value for this argument that will only remove a row if all the values are missing.

Another argument is `axis`. By default this is set to `0` or `index`, which indicates that we are interested in dropping rows. When set to `1` or `columns`, columns will be dropped.

We can constrain which columns to include when checking for missing values, using the `subset` argument.

```
missing_df.dropna(subset=['age'])
```

1 to 25 of 28 entries Filter ?

index	age	salary	previous_company
0	57.0	NaN	1.0
1	56.0	50927.0	NaN
2	46.0	75500.0	3.0
4	60.0	63002.0	1.0
5	54.0	54652.0	NaN
7	64.0	89397.0	3.0
8	60.0	77797.0	4.0
9	61.0	NaN	1.0
10	51.0	92767.0	5.0
12	49.0	91001.0	4.0
13	59.0	54212.0	4.0
14	61.0	84423.0	2.0
16	61.0	84930.0	NaN
17	48.0	63824.0	5.0
18	53.0	53143.0	2.0
19	49.0	59842.0	NaN
20	55.0	79309.0	2.0
21	59.0	NaN	4.0
22	49.0	52895.0	1.0
23	57.0	54035.0	3.0
24	58.0	64962.0	3.0
25	61.0	NaN	1.0
26	63.0	63861.0	2.0
27	47.0	55188.0	3.0
28	63.0	61220.0	3.0

Show 25 per page 1 2

We see that there are still missing data in the *salary* and *previous_company* collumns.

To find out how many rows contain missing data, we can make use of the fact that `True` and `False` are represented by 1 and 0 and can thus be added. The `isna` method will return Boolen values depending on whether the data is missing

```
missing_df.age.isna()
```

```
0     False
1     False
2     False
3      True
4     False
5     False
6      True
7     False
8     False
9     False
10    False
11     True
12    False
13    False
14    False
15     True
16    False
17    False
18    False
19    False
20    False
21    False
22    False
23    False
24    False
25    False
26    False
27    False
28    False
29    False
30    False
31    False
Name: age, dtype: bool
```

We can sum over these Boolean values using the `sum` method. Since `True` values are saved internally to Python as the value 1, the sum will be the number of values marked as `True` when missing, which as we saw, is what the `isna` method returns.

```
missing_df.age.isna().sum()
```

```
4
```

We see that there are 4 missing values in the *age* column.

REPLACING MISSING VALUES

The process of creating values to fill in missing data is called **data imputation** and is a seperate and complicated subject. The pandas library provides a `fillna` method for filling in the missing data with simple calculations.

Below we use the argument and value `method=ffill` which simply fill empty values with previous value. There is also a `method=bfill` argument setting that fills the missing data with the next available data down the column.

```
missing_df.age.fillna(method='ffill')
```

```
0     57.0
1     56.0
2     46.0
3     46.0
4     60.0
5     54.0
6     54.0
7     64.0
8     60.0
9     61.0
10    51.0
11    51.0
12    49.0
13    59.0
14    61.0
15    61.0
16    61.0
17    48.0
18    53.0
19    49.0
20    55.0
21    59.0
22    49.0
23    57.0
24    58.0
25    61.0
26    63.0
27    47.0
28    63.0
29    54.0
30    49.0
31    57.0
Name: age, dtype: float64
```

We can also specify a specific value. For numerical data this could be the median for that variable and for categorical data, it might be the mode. We will learn about summary statistics in the next notebook. For now, we will use the `median` method. It calculate the median for a column with numerical data, ignoring missing data automatically.

```
# The median age (pandas ignores the missing values)
missing_df.age.median()
```

```
57.0
```

We can now impute the missing ages with this median.

```
missing_df.age.fillna(missing_df.age.median())
```

```
0     57.0
1     56.0
2     46.0
3     57.0
4     60.0
5     54.0
6     57.0
7     64.0
8     60.0
9     61.0
10    51.0
11    57.0
12    49.0
13    59.0
14    61.0
15    57.0
16    61.0
17    48.0
18    53.0
19    49.0
20    55.0
21    59.0
22    49.0
23    57.0
24    58.0
25    61.0
26    63.0
27    47.0
28    63.0
29    54.0
30    49.0
31    57.0
Name: age, dtype: float64
```

If we want the changes to be permanent, we have to use the `inplace=True` argument.

DEFAULT MISSING DATA

It is common to use default values when data is not available at the time of capture. If we know what these are, we can interpret them as missing data when the spreadsheet file is imported.

Below, we import a spreadsheet file that uses `999`, `Nil`, and `Missing` for missing values instead of leaving the spreadsheet cell blank.

```
default_missing_df = pd.read_csv('DefaultMissingData.csv')
default_missing_df
```

1 to 25 of 32 entries Filter

index	age	salary	previous_company
0	57	Nil	1
1	56	50927	Missing
2	46	75500	3
3	999	84417	Missing
4	60	63002	1
5	54	54652	Missing
6	999	65739	1
7	64	89397	3
8	60	77797	4
9	61	Nil	1
10	51	92767	5
11	999	59873	2
12	49	91001	4
13	59	54212	4
14	61	84423	2
15	999	Nil	3
16	61	84930	Missing
17	48	63824	5
18	53	53143	2
19	49	59842	Missing
20	55	79309	2
21	59	Nil	4
22	49	52895	1
23	57	54035	3
24	58	64962	3

Show 25 per page 1 2

We can replace the missing values or specify all the words and numbers used for coding missing data when we import the data file.

```
default_missing_df = pd.read_csv('DefaultMissingData.csv', na_values=[999, 'Nil
default_missing_df
```

1 to 25 of 32 entries Filter ?

index	age	salary	previous_company
0	57.0	NaN	1.0
1	56.0	50927.0	NaN
2	46.0	75500.0	3.0
3	NaN	84417.0	NaN
4	60.0	63002.0	1.0
5	54.0	54652.0	NaN
6	NaN	65739.0	1.0
7	64.0	89397.0	3.0
8	60.0	77797.0	4.0
9	61.0	NaN	1.0
10	51.0	92767.0	5.0
11	NaN	59873.0	2.0
12	49.0	91001.0	4.0
13	59.0	54212.0	4.0
14	61.0	84423.0	2.0
17	48.0	63824.0	5.0
21	59.0	NaN	4.0

Those values ar now `NaN`.

WORKING WITH DATES AND TIMES

In this section, we import a new spreadsheet file. It contains data on dates and times of biological laboratory investigations.

```
dt = pd.read_csv('DatesTimes.csv')
dt
```

1 to 24 of 24 entries Filter ?

index	ID	Batch	SpecimenDate	TestDate	TestTime
0	183	3	2025/04/21	2025/04/26	12:23
1	198	2	2025/04/21	2025/04/26	12:26
2	194	1	2025/04/22	2025/04/26	12:11
3	192	2	2025/04/22	2025/04/25	12:05
4	143	2	2025/04/22	2025/04/25	13:45
5	143	2	2025/04/22	2025/04/26	11:59
6	171	2	2025/04/22	2025/04/25	12:34

Let's take a look at the data types.

9	121	3	2025/04/23	2025/04/26	11:34

```
dt.dtypes
```

```
ID                int64
Batch             int64
SpecimenDate     object
TestDate         object
TestTime         object
dtype: object
```

17	190	1	2025/04/26	2025/04/29	12:32

The *SpecimenDate*, *TestDate* and the *TestTime* columns contain objects instead of datetime objects. We can convert these into a proper datetime data type. We will do so by creating a new variable (column header) that combines the two of the columns.

22	157	1	2025/04/28	2025/05/01	10:33

```
dt['DateTime'] = dt.TestDate + ' ' + dt.TestTime  # Add a space
dt
```

1 to 24 of 24 entries Filter ?

index	ID	Batch	SpecimenDate	TestDate	TestTime	DateTime
0	183	3	2025/04/21	2025/04/26	12:23	2025/04/26 12:23
1	198	2	2025/04/21	2025/04/26	12:26	2025/04/26 12:26
2	194	1	2025/04/22	2025/04/26	12:11	2025/04/26 12:11
3	192	2	2025/04/22	2025/04/25	12:05	2025/04/25 12:05
4	143	2	2025/04/22	2025/04/25	13:45	2025/04/25 13:45

This new variable is still an object.

```
dt.DateTime.dtype
```

```
dtype('O')
```

We will now create a new column and use the pandas `to_datetime()` function to convert the object (copied from the *DateTime* column). The `format=` argument allows us to specify the exact format that the object was in. The format of the data in the `DateTime` column is YYYY/MM/DD HH:MM. We use pandas code to indicate these. Uppercase `%Y` specifies the full year, i.e. `2025` instead of just `25`. The rest of the symbols are self explanatory.

18	141	2	2025/04/26	2025/04/30	12:17	2025/04/30 12:17

```
dt['datetime'] = pd.to_datetime(dt.DateTime, format='%Y/%m/%d %H:%M')
dt
```

1 to 24 of 24 entries Filter ?

index	ID	Batch	SpecimenDate	TestDate	TestTime	DateTime	datetime
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

The new `datetime` column is now a datetime object.

2	194	1	2025/04/22	2025/04/26	12:11	2025/04/26 12:11	2025-04-26 12:11:00

```
dt.dtypes
```

```
ID                       int64
Batch                    int64
SpecimenDate            object
TestDate                object
TestTime                object
DateTime                object
datetime        datetime64[ns]
dtype: object
```

12	179	2	2025/04/24	2025/04/29	14:45	2025/04/29 14:45	2025-04-29 14:45:00

Now that this is a datetime object, we might want to analyze this data by month of test. To do so, we create a new column containing the month and use the `dt.month_name` method. We also shorten the month to the first three letters using the `str.slice` method with the `stop` argument set to `3`.

18	141	2	2025/04/26	2025/04/30	12:17	2025/04/30 12:17	2025-04-30 12:17:00

```
dt['month'] = dt.datetime.dt.month_name().str.slice(stop=3)
dt
```

1 to 24 of 24 entries Filter ?

index	ID	Batch	SpecimenDate	TestDate	TestTime	DateTime	datetime	month
						2025/04/26	2025-04-26	
1		2				12:26	12:26:00	

There are various other values, we can extract from the datetime object.

```
dt.datetime.dt.year  # The year
```

```
0     2025
1     2025
2     2025
3     2025
4     2025
5     2025
6     2025
7     2025
8     2025
9     2025
10    2025
11    2025
12    2025
13    2025
14    2025
15    2025
16    2025
17    2025
18    2025
19    2025
20    2025
21    2025
22    2025
23    2025
Name: datetime, dtype: int64
```

```
dt.datetime.dt.hour  # The hour
```

```
0     12
1     12
2     12
3     12
4     13
5     11
6     12
7     13
8     13
9     11
10     9
11    13
12    14
13    15
14    14
15    12
16    13
17    12
18    12
19    11
20    11
21    10
```

```
22    10
23    11
Name: datetime, dtype: int64
```

CONCLUSION

Pandas is a powerful library and we are going to use all the techniques that we have learned about here to manipulate our data in order to do analyses on it.

There is still a lot we have to learn over and above this, though. We will do so, whilest analysing our data.

```
```

✓ 0s completed at 13:52