Python for Subsurface

A basic programming course with applications

Nestor Cardozo
University of Stavanger, Norway

This book was typeset using LaTeX software.

Preface

Python is ranked as the most popular programming language today (Tiobe index) and is the preferred choice for scientists and engineers in computation, data analysis, and machine learning. This is not surprising, as Python has the ideal characteristics of a programming language:

- Simple syntax and readability.
- Comprehensive online help and documentation.
- Built-in mathematical and graphical functions, along with an extensive collection of supplemental libraries.
- A strong community and support system, including abundant tutorials, textbooks, and active forums.
- Open source and freely available.
- Compatible across all major platforms: Windows, macOS and Linux.
- Runs efficiently even on modest hardware (e.g., Raspberry Pi).

Python is well-suited for analyzing and visualizing subsurface data. Well and seismic data can be easily imported into powerful data structures such as dataframes and arrays, facilitating data manipulation, analysis, and visualization.

This course provides a hands-on introduction to Python with practical applications in subsurface studies. It is designed for geoscientists and engineers who work daily with subsurface data but may have little or no programming experience.

Can one learn Python in just a few days and apply it to meaningful subsurface tasks? The answer is yes. With the rise of AI-powered code assistants and Python's extensive library ecosystem, learning syntax becomes less critical than understanding fundamental programming concepts and logical problem-solving.

This course follows that approach. Examples from geosciences and reservoir engineering illustrate both Python fundamentals (Chapters 2–5) and more advanced topics, such as data visualization and analysis (Chapters 6–7).

I hope this course inspires you to use Python in your daily work and encourage you to continue learning on your own.

Acknowledgements

These notes are adapted from a one-week course I taught at ConocoPhillips in Stavanger in May 2025. I would like to thank Mathias Tomasgaard, Karl Simpson, and Dylan Loss for organizing the course. The original material was based on internal ConocoPhillips data, which cannot be shared publicly. To make the course more broadly accessible, I have revised the content to incorporate publicly available datasets from three sources:

- Well log data from the FORCE 2020 Machine Learning competition.
- the F3 seismic dataset provided by TerraNubis.
- The FactPages database from the Norwegian Offshore Directorate

Thanks to these organizations for making these data freely available to the community.

Contents

Chapter 1

Getting started

There are mainly two strategies to setup a Python environment suitable for scientific computing on your computer:

1. Install the Python core and add all the required scientific packages separately.
2. Install a "ready-to-use" Python environment, specifically developed for scientific purposes.

We will choose option 2 because it requires no programming skills and we will be ready to start coding.

1.1 Installing Anaconda

An example of a "ready to use" scientific Python environment is the Anaconda Python Distribution. Anaconda is a free Python science distribution that contains:

- conda - a package and environment manager for the command line interface.
- Anaconda Navigator - a desktop application with options to launch applications and manage environments.
- over 250 scientific and machine learning packages. This is perhaps more than we need.

Download Anaconda. You can skip the registration and follow the steps to install it on your computer.[1]

[1] Depending on your organization's setup, you may need to use the Anaconda distribution on your server.

Once Anaconda is installed, you can open the Anaconda Prompt from the Search box (Windows) or a Terminal window (macOS or Linux; Figure 1.1a). From the Search box you can open as well Anaconda Navigator (Windows), or in the Applications folder double-click the Anaconda Navigator icon (macOS; Figure 1.1b).

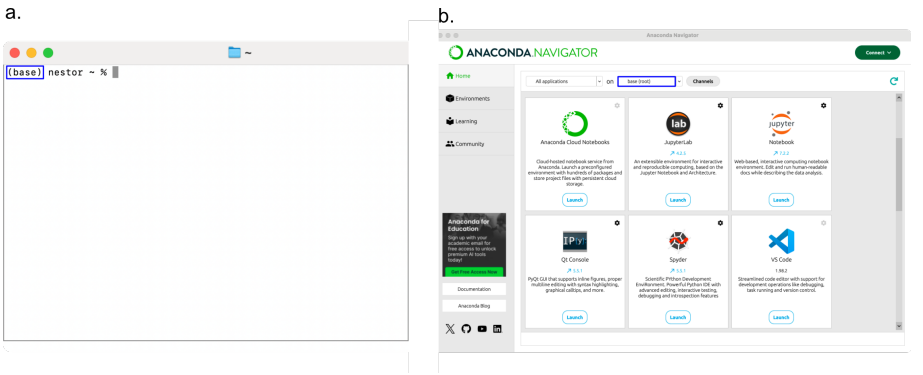


Figure 1.1: **a.** Command prompt, and **b.** Anaconda Navigator Home page.

The default environment of Anaconda is called `base` (Figure 1.1). To get a list of the packages that are installed in this environment, type in the command prompt:

```
conda list
```

Alternatively, in Anaconda Navigator choose the Environments page. The packages installed in the `base` environment are listed there. You can think of the Anaconda Prompt as the command line interface, while Anaconda Navigator is the graphical user interface (GUI).

1.2 Where to write Python code

After installing Python, you can write code practically anywhere, in a text editor or the terminal, but in this course we will use the program VS Code to do that:

1. Download VS Code from this website, and install it on your computer.
2. Open VS Code. You will get a window like Figure 1.2a.
3. Although you have already installed Python, VSCode does not have a Python interpreter yet. Therefore, you need to install a Python extension. On the side bar (activity bar), select the Extensions icon (red rectangle in Figure 1.2a). You will be presented with a list of extensions (Figure 1.2b).

4. Install the Python extension by clicking its Install button.
5. Install the Jupyter extension by clicking its Install button. This will allow you to work with Jupyter notebooks.

That's it. You are ready to write code.

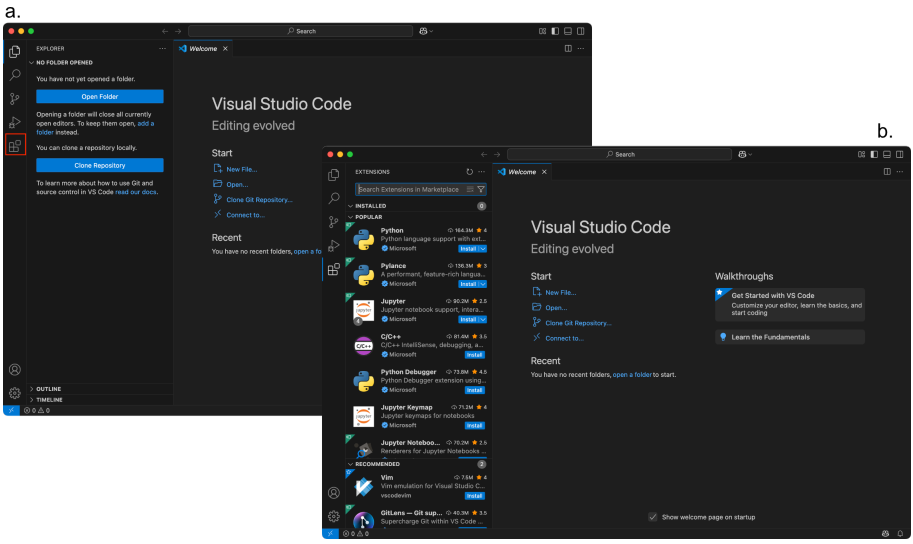


Figure 1.2: **a.** VS Code Welcome window. Red square is the Extensions icon. **b.** VS Code extensions, including Python and Jupyter.

1.2.1 Jupyter Notebooks

Jupyter Notebooks are files with extension `.ipynb` which can combine code, text, figures and interactive controls. To work with notebooks, you can use VS Code (since the Jupyter extension is installed)[2]:

1. Select the Explorer icon in the activity bar (Figure 1.3a).
2. Open a folder. In Figure 1.3a, I selected the folder PFS. If asked for it, say that you thrust the authors of the folder.
3. Expand the PFS folder (Select the > sign), and select the New File icon (document with a +). Name the file `my_first_notebook.ipynb`.
4. Click the Select Kernel icon and choose Python Environments -> base environment. After this, the Kernel should say `base`.
5. Add a code cell to the notebook (+ Code).

[2]Some notebooks may not work well in VS Code. If that is the case, we can use JupyterLab.

6. In the code cell, type these two lines :

```
a = "Welcome to Python for Subsurface"
print(a)
```

7. Select the right pointing triangle to the left of the cell to run it. You should get a result like in Figure 1.3a.

8. Save the notebook.

Congratulations, you have made your first notebook!

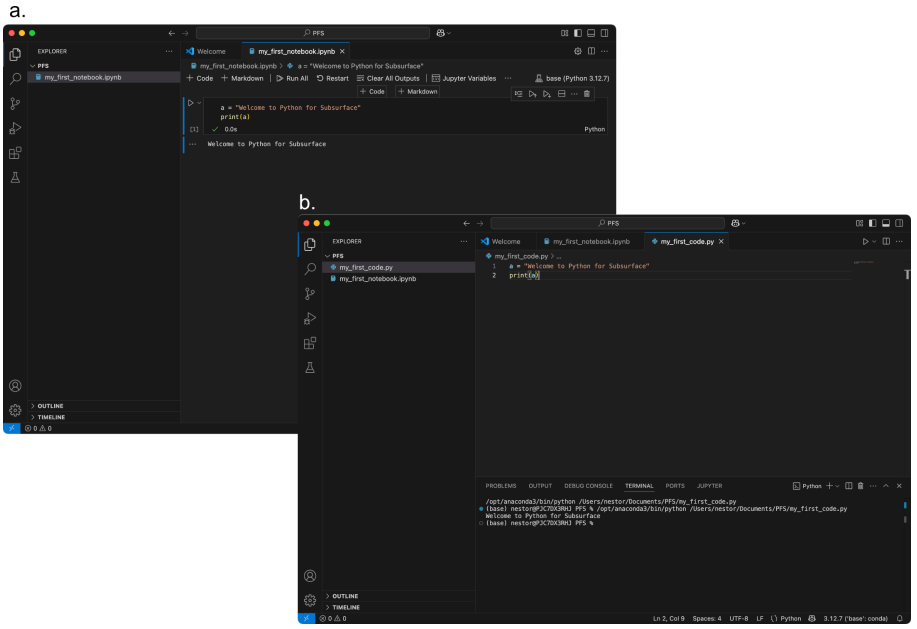


Figure 1.3: **a.** Jupyter notebook and **b.** Python file in VS Code.

1.2.2 Python files

Notebooks are great for teaching and showing the sequential execution of code. However, they are not good for writing and debugging programs, or structuring code. For that, we need Python files. Let's write our first Python file in VSCode:

1. Create a new file and name it `my_first_code.py` (Figure 1.3b).

2. In the lower right corner of the window, make sure the `base` environment is chosen.

3. In the file, type the same code as above.
4. Save the file.
5. Run the file by selecting the right pointing triangle in the upper right corner of the window.
6. The output will be presented in a Terminal view, like in Figure 1.3b.

Notice that Python (`.py`) files are simpler than notebook (`.ipynb`) files, and they can be edited on any text program.

1.3 Managing conda environments

The Anaconda `base` environment houses conda and it needs to be protected to keep conda functioning properly. Therefore, every time you start a new project that requires some special packages, it is recommended to create a new environment. Let's create a new environment for this course:

1. Launch the Anaconda Prompt.
2. Type:

```
conda env list
```

   This will output a list of the installed environments.

3. Type:

```
conda create --name pfs
```

   This creates a new environment called `pfs` (Python for Subsurface).

4. Type:

```
conda env list
```

   to check the environment is installed. You should see the `base` and `pfs` environments.

5. Activate the new environment by typing:

```
conda activate pfs
```

6. To see the packages installed in the environment, type:

```
conda list
```

   As you can see, there are no packages installed yet.

7. Now install the packages you need for the course. Type the following:

```
conda install numpy pandas matplotlib plotly hvplot \
seaborn scipy scikit-learn jupyterlab ipywidgets
```

   For formatting reasons, I broke the line (\), but you can have everything in one line.

8. You will also need to install another package using `pip`. Type the following:

```
pip install segyio
```

9. Type again:

```
conda list
```

   to check the packages installed.

10. To deactivate the environment and come back to the `base` environment, type:

```
conda deactivate
```

That's it! You've now set up the environment for the course, and you can select it in VS Code as we discussed in section 1.2. For further information check this cheat sheet for managing conda environments.

Note

As shown above, a package can also be installed using the command `pip install`. However, always try `conda install` first, because conda is better at checking for internal consistency between the packages. If that does not work, then you can try `pip install`. This will work most of the time.

1.4 Online resources

There are many online resources to learn Python. For this course, I recommend the following:

- w3schools offers clear tutorials, interactive examples, and quick references for learning Python.
- Real Python offers in-depth tutorials and coding exercises for a more detailed understanding of Python.
- GitHub provides access to many open-source Python repositories, where one can explore real-world projects, contribute, and learn from experienced developers. I do recommend creating a GitHub account.
- The NumPy user guide is a great resource for learning NumPy arrays. A good place to start is the absolute basics for beginners
- Finally, Python for Data Analysis is the ultimate guide to mastering Pandas and harnessing the power of data analysis in Python.

1.5 AI code assistants

AI code assistants have transformed programming by shifting the focus from syntax to logic and structure (Figure 1.4). There are several AI assistants you can use:

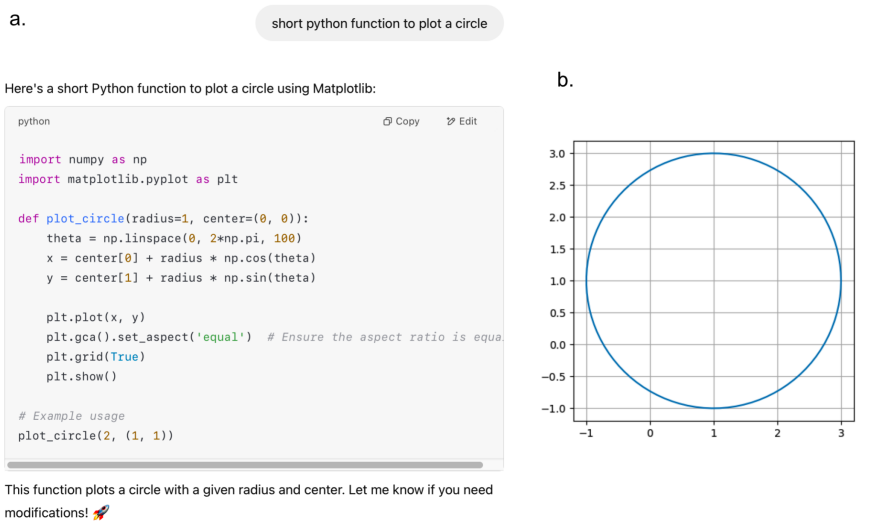


Figure 1.4: **a.** ChatGPT query, and **b.** the result of running the code.

- Microsoft Copilot.
- ChatGPT.
- GitHub Copilot. If you are using this assistant, make sure to install in VS Code the GitHub Copilot extension. After that and signing to your GitHub account, you will be able to use GitHub Copilot directly in VS Code.

I would encourage you to use AI assistants in your coding. The code generated is generally good, and if there are parts you don't understand, it is possible to get further help from the assistant. This book teaches Python using GitHub Copilot and ChatGPT.

1.6 Exercises

1. Use Copilot or ChatGPT to learn how to plot the sine function in Python. Ask the AI assistant to explain what the suggested code does. Then, copy the code into both a Jupyter Notebook (`.ipynb`) and a standard Python script file (`.py`) using VS Code. Run the code in both environments and verify that the output is correct.
2. Next, plot both the sine and cosine functions in Python, with one graph positioned above the other in the same figure. Repeat the same steps as in the previous task: use the AI assistant, review the explanation, add the code to both file formats, and confirm that the output displays as expected.

Chapter 2

Data types

Coding is mostly about working with information. In Python, we use *variables* to store and pass that information. A variable is just a name that holds data like a number or text. Python automatically sets aside memory based on the value you assign. You don't need to declare the type—just use the `=` sign, with the variable on the left and the value on the right. Python will figure out the data type for you. For example[1]:

```
# Creating two variables
age = 65
top = "Cretaceous"
```

Here, `65` and `"Cretaceous"` are assigned to the variables `age` and `top` respectively. The first line starting with `#` is a comment. Comments are ignored by Python but are useful for documenting your code. Including comments is always a good practice, as it helps others (and your future self) understand what your code is doing.

Figure 2.1 shows the standard data types in Python. These are covered in detail in the next sections, starting with Numbers.

2.1 Numbers

Python supports integers, floats, and complex numbers. Integers and floats differ by the presence or absence of decimals. Complex numbers have a real part and an imaginary part:

[1]The code for this chapter is included in the notebook `chapter2.ipynb`. As a best practice, I recommend setting the kernel in VS Code to the course environment for all notebooks and Python files throughout the course.

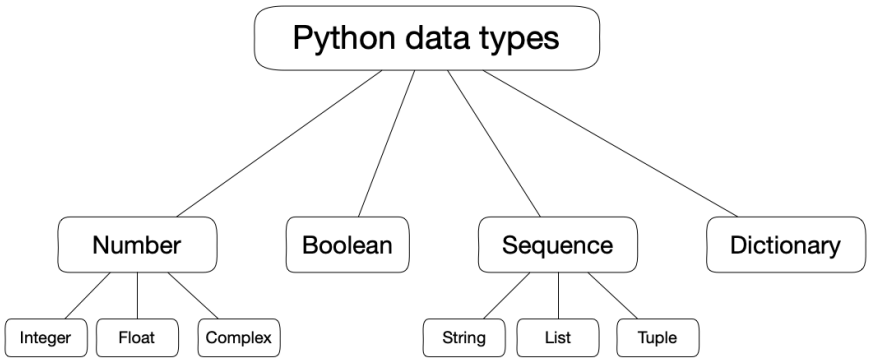


Figure 2.1: Standard data types in Python

```
my_integer = 32 # integer
my_float = 23.00 # float
my_complex = 5 + 3j # complex: real part = 5, imaginary part = 3
```

Python has several built-in functions. We can use the function `type()` to find out the type of a variable, and the function `print()` to output this information:

```
print(type(my_integer), type(my_float), type(my_complex))
```

```
<class 'int'> <class 'float'> <class 'complex'>
```

So our numbers are actually *classes*. The concept of class comes from the object oriented philosophy of Python (chapter 5). For the moment, just realize that the numbers above are *class instances* or *objects* that come with built-in *members*. Some of these members are *attributes* of the object. Others are *methods*, which perform actions and must be called using parentheses `()`. For example, to output the real and imaginary parts of `my_complex`, we can do the following:

```
print(my_complex.real, my_complex.imag)
```

```
5.0 3.0
```

Here, we use the attributes `real` and `imaginary` of `my_complex` to extract its real and imaginary parts.

When you write code in VS Code, the editor suggests completions as you type. You can press the `Tab` key to insert the most relevant suggestion. This functionality is powered by *IntelliSense*, which includes code completion, content assist, and code hinting. For example, if you type `my_complex.r`, the program will show a list of members of `my_complex` starting with `r`, with the best match—`real`—at the top. You can see quick info for each member by selecting the > icon, which expands the accompanying documentation to the side 2.2). IntelliSense is a powerful tool that helps you explore Python functions, variables, and object members—such as attributes and methods—as you write code. For more information, check out this site.

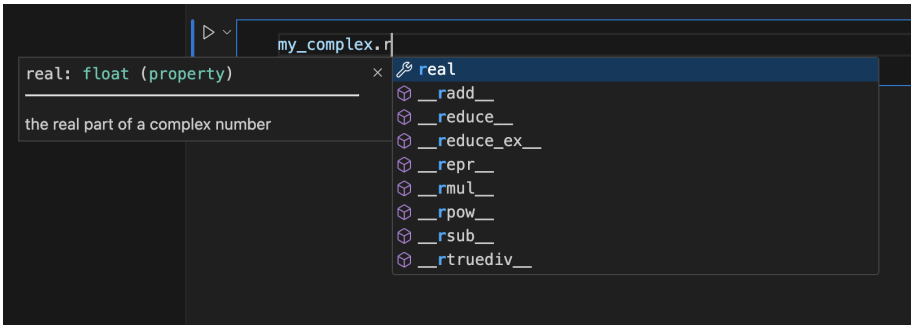


Figure 2.2: IntelliSense in VS Code.

Alternatively, you can use Python's dir() function to view all the attributes and methods of an object.

2.1.1 Mathematical operations

Python can be used as a simple calculator. For example:

```
a = 27
b = 4
print(a + b) # addition
print(a - b) # subtraction
print(a * b) # multiplication
print(a / b) # division
print(b ** 3) # exponentiation
print(a // b) # floor division
print(a % b) # modulo division
```

```
31
23
108
```

```
6.75
64
6
3
```

Notice that division `/` returns a float value, even though the two numbers are integers. Floor (or integer) division `//` returns the quotient of the division rounded down to the nearest whole number, while modulo division `%` returns the remainder of the division (for negative numbers this is more complicated, see this note).

These operations also work with complex numbers. Let's try the following:

$$\frac{3i}{4-3i} = -\frac{9}{25} + \frac{12}{25}i \approx -0.360 + 0.480i \tag{2.1}$$

In Python this is easy:

```
z0 = 3j # complex, imaginary part = 3
z1 = 4 - 3j # complex, real part = 4, imaginary part = -3
z2 = z0/z1 # division of complex numbers
print(z2.real, z2.imag) # output the result
```

```
-0.36 0.48
```

However, for more advanced operations, we need to import the `math` library:

```
import math # import math library

angle = 30 # angle in degrees
# sine of angle, option 1
sine_one = math.sin(30*math.pi/180)
# sine of angle, option 2
sine_two = math.sin(math.radians(angle))
# output sine
print(sine_one, sine_two)
```

```
0.49999999999999994 0.49999999999999994
```

The first line of the code imports the math library, giving access to its functions using the `math.function_name()` format. Options 1 and 2 both calculate the sine of an angle. Note that the angle must first be converted from degrees to radians, since Python (and most programming languages) use radians by default. Finally, the result isn't exactly `0.5` due to floating-point precision errors—these small inaccuracies are a normal consequence of how computers

represent decimal numbers.

The `math` library has a wide range of mathematical functions. You can find the complete list here. Another way to quickly find out the functions in the math library is to type `math.`; VS Code will show a list of the functions, and you can find information about any of them by selecting the > icon.

2.2 Booleans

Booleans (or bools) can only have two values, `True` or `False`. Let's look at some examples:

```
a = 10 # variable a
b = 7 # variable b

c = (a > b)  # bool c
d = (a == b) # bool d
e = (a < b)  # bool e
f = (a != b)  # bool f

print(c, d, e, f) # output bools
```

```
True False False True
```

The operators `>` (greater than), `==` (equal), `<` (lower than), and `!=` (not equal) are called comparison operators. You can find a list of these operators here.

Python has a nice feature: When a bool is used in an arithmetic expression, Python translates a `True` to one and `False` to zero. This is useful if we want to assign different numerical values to a variable based upon the value of a bool. For example, suppose we have a compass that cannot accurately measure any angle below 1°. Thus, any measurement less than 1° should be 0°. We could do the following:

```
angle = 0.5
corrected_angle = (angle >= 1.0) * angle
print(corrected_angle)
```

```
0.0
```

Let's wrap up this section with a mind-blowing experiment. Can you predict what the output of the following code will be? Go ahead and run it to see the result—then try to figure out why it behaves that way. Hint: Try printing the values of `a` and `b` to help you understand what's going on.

```
a = 0.1 + 0.2
b = 0.3
print(a == b)
```

2.3 Sequences

A sequence is an ordered collection of items. Examples of sequences are strings, lists, and tuples. Strings are sequences of characters, lists are ordered collections of data (which can be of different type), and tuples are similar to lists, but they cannot be modified after their creation. The elements of a sequence can be accessed using indexes within brackets (`[]`):

- The first index of a sequence is always 0.
- Using negative integers (e.g., -1) as indices accesses elements starting from the end of the sequence, where -1 refers to the last element, -2 to the second last, and so on.
- Two integers separated by a colon (e.g. 3:7) define an index range, sampling the sequence from the lower to the upper index, but **not including** the upper index (i.e., indexes 3, 4, 5 and 6).
- Omitting the lower index in a slice (e.g., :5) means slicing from the start of the sequence up to, but not including, the upper index.
- Omitting the upper index in a slice (e.g., 4:) means slicing from the lower index through to the end of the sequence.
- Negative indices can also be used in slices (e.g., -2:), which selects elements starting from the second-to-last element through to the end of the sequence.
- A slice can include a third value, called the step (e.g., 1:10:2), which defines the stride between elements selected (in this example, every second element).
- Negative steps in a slice (e.g., ::-1) reverse the sequence by stepping backward through it.

Let's look now at each of the sequence types, starting with strings.

2.3.1 Strings

Strings are sequences of characters and are defined within quotes—either single quotes (`'hello'`), or double quotes (`"hello"`). Personally, I prefer using double quotes since a string may contain an apostrophe (`'`). Strings can be concatenated with the `+` operator and repeated with the `*` operator. Let's take a look:

```
my_string = "Stressed " + "desserts" # string concatenation
print(my_string) # output string
print(my_string * 2) # repeat string
print(my_string[0:6:1]) # print characters: 0 to 6-1 with step 1
print(my_string[:6]) # print first 6 characters
print(my_string[-8:]) # print last 8 characters
print(my_string[::2]) # print every second character
print(my_string[::-1]) # print string in reverse order
```

```
Stressed desserts
Stressed dessertsStressed desserts
Stress
Stress
desserts
Srse esrs
stressed dessertS
```

Notice that the blank space is also a character. Formatted strings (f-strings) can be used to include numerical values with a desired notation. Here is one example:

```
r  = 6371 # Earth radius in km
V  = 4/3 * math.pi * r**3 # Earth volume in km^3

# output Earth volume using f-string
print(f"Earth volume is {V:.2f} km^3") # 2 decimals
print(f"Earth volume is {V:.0f} km^3") # no decimals
print(f"Earth volume is {V:.4e} km^3") # scientific, 4 decimals
```

```
Earth volume is 1083206916845.75 km^3
Earth volume is 1083206916846 km^3
Earth volume is 1.0832e+12 km^3
```

Note that the string is prefixed with an `f`, and the value goes inside curly braces `{}`, with a format specifier for the desired notation. f-strings were introduced in Python 3.6 and they are the preferred way of formatting strings. You can find more information about them here.

To finalize, it's important to note that strings in Python are *immutable*. This means you cannot change their content after creation. For example, the following code will raise an error:

```
s = "Fake"
s[0] = "R" # error: string does not support item assignment
```

Instead, you should do something like:

```
s = "Fake"
s = "R" + s[1:] # replace F with R
```

or:

```
s = "Fake"
s = s.replace("F", "R") # replace F with R
```

2.3.2 Lists

Lists are perhaps the most flexible type of sequence in Python, as they can contain different data types (e.g., numbers and strings) and can grow or shrink in size to accommodate their elements. Whenever you need to collect multiple values in Python, lists are a great option. Lists are defined inside squared brackets `[]`. The symbol `*` can be used to create copies of a list, and the symbol `+` can be used to concatenate lists. The function `len()` returns the number of elements in a list (and any collection). For example:

```
# empty list
empty = []
print(empty)

# list made by copying and concatenating elements
zero = [0, "zero"] # list with integer and string
one = [1, "one"] # list with integer and string
my_list = zero * 2 + one * 2 # list with 2 zeros and 2 ones
print(my_list)

# number of elements in list
print(len(my_list))
```

```
[]
[0, 'zero', 0, 'zero', 1, 'one', 1, 'one']
8
```

Lists have methods to add, insert or delete elements:

```
tops = [252 ,"TR", 200, "J", 145, "K", 66, "Pa"] # geologic tops
print(tops)

tops.insert(0, "P") # insert Permian at index 0
tops.append(23) # append 23
tops.extend(["N", 2.6, "Q", 0.0, "A"]) # add another list
print(tops)

tops.pop(-1) # remove Anthropocene
print(tops)

tops.reverse() # reverse the list
print(tops)
```

```
[252, 'TR', 200, 'J', 145, 'K', 66, 'Pa']
['P', 252, 'TR', 200, 'J', 145, 'K', 66, 'Pa', 23, 'N', 2.6, 'Q', 0.0, 'A']
['P', 252, 'TR', 200, 'J', 145, 'K', 66, 'Pa', 23, 'N', 2.6, 'Q', 0.0]
[0.0, 'Q', 2.6, 'N', 23, 'Pa', 66, 'K', 145, 'J', 200, 'TR', 252, 'P']
```

And of course, you can access the elements of the list by their index:

```
print(f"The oldest top is {tops[-1]} and it is {tops[-2]} My")
```

```
The oldest top is P and it is 252 My
```

2.3.3 Tuples

A tuple is also an ordered collection of elements which can be of different type. However, a tuple is *immutable*. Once you create a tuple, it cannot be modified. Tuples are defined inside parentheses `()`:

```
f_tops = (252 ,"TR", 201, "J", 145, "K", 66, "Pa") # geologic tops
print(f_tops)
```

```
(252, 'TR', 201, 'J', 145, 'K', 66, 'Pa')
```

The following code will throw an error:

```
f_tops[2] = 200 # error: tuple does not support item assignment
```

Tuples are useful for storing data that should remain constant throughout the execution of a program—for example, the months of the year.

2.4 Dictionaries

Dictionaries are collections of key-value pairs. They are defined inside curly braces `{}`, with each key-value pair separated by a comma, and each key separated from its value by a colon. The order of items in a dictionary does not matter. To retrieve a value, you use the corresponding key inside square brackets.

Let's update the code we wrote earlier to calculate the volume of the Earth (p. 14), and make it more flexible by calculating the volume ratio of any two planets in the solar system. To do this, we'll use a dictionary, with the keys being the planets' names, and the values the planets' radii:

```
# dictionary with planets in the solar system and their radii
planets = {"Earth": 6371, "Mars": 3390, "Venus": 6052,
           "Mercury": 2439, "Jupiter": 69911, "Saturn": 58232,
           "Uranus": 25362, "Neptune": 24622}

# Set the two planets names here
p_1 = "Earth"
p_2 = "Mercury"

r_1 = planets[p_1] # radius of planet 1 in km
r_2 = planets[p_2] # radius of planet 2 in km


v_1 = 4/3 * math.pi * r_1**3 # volume of planet 1 in km^3
v_2 = 4/3 * math.pi * r_2**3 # volume of planet 2 in km^3
v_ratio = v_1 / v_2 # volume ratio of the two planets

# output volume ratio
print(f"{p_1} volume is {v_ratio:.2f} larger than {p_2} volume")
```

```
Earth volume is 17.82 larger than Mercury volume
```

It is easy to add and remove key-value pairs from a dictionary. The dictionary has also methods to list its keys, values, or items:

```
# Define dictionary of mineral hardness
mohs = {"talc":1, "gypsum":2, "calcite":3, "fluorite":4,
        "apatite":5, "orthoclase":6, "quartz":7,
        "topaz":8, "corundum":9, "diamond":10}

# adding key-value pairs to dictionary
mohs["olivine"] = 6.5
mohs["salt"] = 2.5
mohs["pyrite"] = 6.25

# delete pyrite from dictionary
del mohs["pyrite"]

print(mohs.keys(), "\n") # print dictionary keys
print(mohs.values(), "\n") # print dictionary values
print(mohs.items(), "\n") # print dictionary items
print("Hardness of salt =", mohs["salt"]) # print hardness of salt
```

```
dict_keys(['talc', 'gypsum', 'calcite', 'fluorite', 'apatite', 'orthoclase', '
    quartz', 'topaz', 'corundum', 'diamond', 'olivine', 'salt'])

dict_values([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6.5, 2.5])

dict_items([('talc', 1), ('gypsum', 2), ('calcite', 3), ('fluorite', 4), ('
    apatite', 5), ('orthoclase', 6), ('quartz', 7), ('topaz', 8), ('corundum',
    9), ('diamond', 10), ('olivine', 6.5), ('salt', 2.5)])

Hardness of salt = 2.5
```

Dictionaries offer a flexible and efficient way to organize data that maps unique keys to specific values, making them ideal for storing information like petrophysical properties or unit conversion factors. JSON files are basically just Python dictionaries, which shows how important dictionaries are.

2.5 Exercises

1. The average radius of the Earth is 6371 km. The oceans cover 70% of the surface of the planet, and the average depth of the oceans is 3700 m. Estimate the surface area of the Earth, the surface area occupied by oceans, and the volume of the oceans (Hint: The surface area of a sphere is $4\pi r^2$).

2. Update the code to calculate the volume ratio between any two planets in the solar system (page 18), so that the user can enter the names of the two planets interactively. Hint: Use the Python `input()` function.

3. The table below shows the conversion of different pressure units to Pascals (Pa):

Unit	**Pa**
atm	101,325
bar	100,000
MPa	1'000,000
mm Hg	133.32
mm H_2O	9.81
psi	6894.76

   (a) Make a dictionary to represent this table. The keys of the dictionary are the units names, and the values are the units in Pascals.

   (b) The maximum depth of the 14.4 km long Ryfylke tunnel is 292 m below sea level. What is the pressure at this depth in Pa? Hint: Pressure = water density * gravity * depth. Use water density = 1000 kg/m^3, and gravity = 9.8 m/s^2.

   (c) What is the pressure at this depth in atm, mm Hg, and psi? Use your dictionary to solve this problem.

Chapter 3

Enhanced data structures

In Chapter 2, we covered Python's basic data types. For more complex and efficient data analysis, however, we often rely on arrays and DataFrames. These are powerful data structures offered by the NumPy and Pandas libraries, respectively.

3.1 Arrays

Like lists, arrays are collections of items, but of the same type (e.g., all numbers or all strings). To use arrays, we need to import the NumPy library[1]:

```
import numpy as np # import numpy library under the alias np
```

However, contrary to lists, arrays need to be declared:

```
my_array = np.array([-10, 30, 60, 90, 120, 150, -100]) # new array
print(my_array)
```

```
[ -10   30   60   90  120  150 -100]
```

Let's modify the array:

```
my_array[0] = 0 # modify first element
my_array[-1] = 180 # modify last element
print(my_array)

my_array = np.append(my_array,[240, 270, 300, 330, 360]) # append
print(my_array)
```

[1]The code for this section is included in the notebook `chapter3_1.ipynb`

```
my_array = np.insert(my_array,7,210) # insert element at index 7
print(my_array)

my_array = np.delete(my_array,-1) # delete last element
print(my_array)
```

```
[  0  30  60  90 120 150 180]
[  0  30  60  90 120 150 180 240 270 300 330 360]
[  0  30  60  90 120 150 180 210 240 270 300 330 360]
[  0  30  60  90 120 150 180 210 240 270 300 330]
```

Notice that arrays do not have methods to append, insert, or delete elements. Instead, we use NumPy's standalone `append()`, `insert()`, or `delete()` methods, each of which returns a new array. This means arrays are not ideal for collecting objects incrementally—every time we add an item, a new array is created. If you're adding many items, this repeated copying can significantly slow down your code.

Now, let's use index ranges to print some elements of the array:

```
print(my_array[4:8]) # print elements with indexes 4 to 7
```

```
[120 150 180 210]
```

Index ranges are quite powerful. Suppose we want to calculate the differences between successive elements of the array. We can do this in one line of code as follows:

```
# differences between succesive elements of my_array
diffs = my_array[1:] - my_array[:-1]
print(diffs)
```

```
[30 30 30 30 30 30 30 30 30 30 30]
```

`my_array[1:]` contains the second to the last element of the array, while `my_array[:-1]` contains the first to the penultimate element of the array. Subtracting these two arrays gives us the differences between the elements of the array.[2]

We can use the array `size` attribute to get the number of elements in the array:

```
print("number of elements in array =", my_array.size)
```

[2]This can also be accomplished using the NumPy `diff()` method.

```
number of elements in array = 12
```

and the `dtype` attribute to find out the type of elements in the array:

```
my_array.dtype
```

```
dtype('int64')
```

3.1.1 2D arrays

A 2D array is an array of 1D arrays. It can be constructed as follows:

```
# create a 3 x 4 array
my_2d_array = np.array([[1, 2, 3, 4], [5, 6, 7, 8],
                        [9, 10, 11, 12]])
print(my_2d_array)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

To access an element of the array, we use two indexes within brackets. The first index refers to the row, and the second index to the column of the array. This is illustrated in Figure 3.1 with a library cabinet for the box at row index 2 and column index 2:

```
# element in third row and third column
print(my_2d_array[2,2])
```

```
11
```

Index ranges allow us to quickly access several elements of the array. This is referred to as *slicing* the array. Figure 3.2a shows how to access the second row of the cabinet. Let's do the same with our array:

```
# second row of my_2d_array
print(my_2d_array[1,:]) # my_2d_array[1] does the same
```

```
[5 6 7 8]
```

And Figure 3.2b shows how to select the second column of the cabinet. Let's do the same with our array:

Figure 3.1: A library cabinet is a 2D array.

```
# second column of my_2d_array
print(my_2d_array[:,1]) # : means all rows
```

```
[ 2  6 10]
```

We can of course select several rows or columns at once:

```
# first two rows of my_2d_array
print(my_2d_array[:2,:], "\n") # my_2d_array[:2] does the same

# last two columns of my_2d_array
print(my_2d_array[:,2:])
```

```
[[1 2 3 4]
 [5 6 7 8]]

[[ 3  4]
 [ 7  8]
 [11 12]]
```

We can use the array `shape` attribute to obtain the number of rows and columns in the array. This returns a tuple whose first element is the number of rows, and second element is the number of columns:

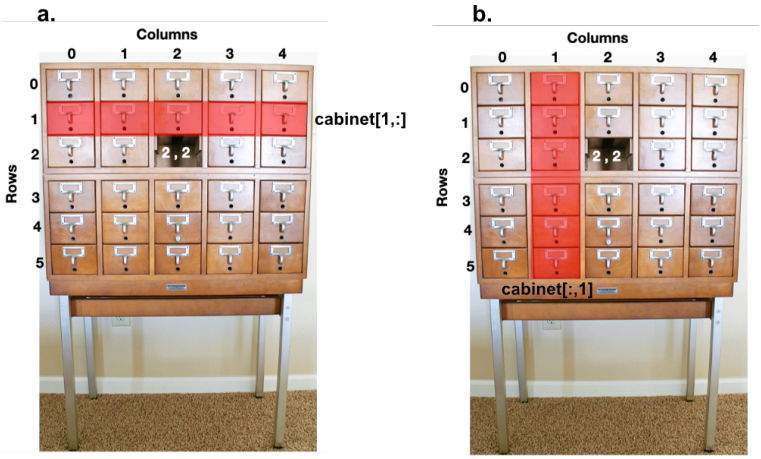


Figure 3.2: Selecting **a.** the second row, and **b.** second column of cabinet.

```
# number of rows in my_2d_array
print("number of rows in array =", my_2d_array.shape[0])

# number of columns in my_2d_array
print("number of columns in array =", my_2d_array.shape[1])
```

```
number of rows in array = 3
number of columns in array = 4
```

3.1.2 3D arrays

3D arrays work the same way, they are arrays of 2D arrays:

```
my_3d_array = np.arange(24).reshape(2,3,4) # 2 x 3 x 4 array
print(my_3d_array)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

In the code above, the NumPy `arange()` method generates a 1D array of integers from 0 to 23. The array `reshape()` method, reshapes the array into a 3D array of two 2D arrays, each one with 3 rows and four columns. We can use indexes to slice the array:

```
print(my_3d_array[0], "\n") # first 2D array in 3D array
print(my_3d_array[1], "\n") # second 2D array in 3D array
print(my_3d_array[0,1], "\n") # second row of first 2D array
print(my_3d_array[1,:,2], "\n") # third column of second 2D array
print("shape of array", my_3d_array.shape) # shape of 3D array
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]

[4 5 6 7]

[14 18 22]

shape of array (2, 3, 4)
```

Finally, the array `ndim` attribute tells us the dimensions of the array:

```
print(my_array.ndim)
print(my_2d_array.ndim)
print(my_3d_array.ndim)
```

```
1
2
3
```

3.1.3 Boolean arrays

Comparison operators also work on arrays. Suppose we have two arrays with the months of the year, and their precipitation values in mm:

```
months = np.array(["Jan", "Feb", "Mar", "Apr", "May", "Jun",
                   "Jul", "Aug", "Sept", "Oct", "Nov", "Dec"])
precip = np.array([142, 89, 114, 74, 53, 38,
                   13, 25, 43, 109, 165, 137])
```

Suppose we want to extract the months with precipitation higher than 100 mm. We can do the following:

```
high_precip = precip > 100

print(high_precip, "\n") # output array

# Output months with precipitation > 100 mm
print("Months with precipitation > 100 mm:", months[high_precip])
print("Precipitation in these months:", precip[high_precip])
```

```
[ True False  True False False False False False False  True  True  True]

Months with precipitation > 100 mm: ['Jan' 'Mar' 'Oct' 'Nov' 'Dec']
Precipitation in these months: [142 114 109 165 137]
```

`high_precip` is a Boolean array which we use to extract the high precipitation elements from the `months` and `precip` arrays. Wherever `high_precip` is `True` the elements of these arrays will be returned. This is a very efficient way to filter arrays (and in fact any collection).

3.1.4 Array operations

There are two main groups of operations that involve NumPy arrays:

- Element-wise operations
- Linear algebra operations

Element-wise operations

These are simple element-wise operations that involve an array[3], and array and a scalar, or two arrays of the same dimension. For example:

```
print(np.around(np.sin(np.radians(my_array)),2)) # sine of array
print(np.around(np.cos(np.radians(my_array)),2)) # cosine of array
```

```
[ 0.    0.5   0.87  1.    0.87  0.5   0.   -0.5  -0.87 -1.   -0.87 -0.5 ]
[ 1.    0.87  0.5   0.   -0.5  -0.87 -1.   -0.87 -0.5  -0.    0.5   0.87]
```

```
# create a 3 x 3 array
array_a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(array_a, "\n") # print array_a

print(array_a + 2, "\n") # array plus scalar
print(array_a - 2, "\n") # array minus scalar
print(array_a * 2, "\n") # array times scalar
print(array_a / 2, "\n") # array divided by scalar
print(array_a ** 2) # array elevated to the scalar
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

[3] Mathematical functions on an array, and NumPy has plenty of them.

```
[[-1  0  1]
 [ 2  3  4]
 [ 5  6  7]]

[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]

[[0.5 1.  1.5]
 [2.  2.5 3. ]
 [3.5 4.  4.5]]

[[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

```
# Create another 3 x 3 array
array_b = np.array([[9, 8, 7], [6, 5, 4], [3, 2, 1]])

print(array_a + array_b, "\n") # element-wise sum
print(array_a - array_b, "\n") # element-wise difference
print(array_a * array_b, "\n") # element-wise multiplication
print(array_a / array_b, "\n") # element-wise division
print(array_a ** array_b) # element-wise exponentiation
```

```
[[10 10 10]
 [10 10 10]
 [10 10 10]]

[[-8 -6 -4]
 [-2  0  2]
 [ 4  6  8]]

[[ 9 16 21]
 [24 25 24]
 [21 16  9]]

[[0.11111111 0.25       0.42857143]
 [0.66666667 1.         1.5       ]
 [2.33333333 4.         9.        ]]

[[   1  256 2187]
 [4096 3125 1296]
 [ 343   64    9]]
```

Processing all elements of an array simultaneously can significantly speed up your code—a technique known as *vectorization*. Whenever possible, you should aim to vectorize your operations for better performance.

Linear algebra operations

The NumPy library is an excellent tool for performing linear algebra operations. Let's explore a few examples:

```
# create two vectors (1 x 3 arrays)
vector_u = np.array([1, 2, 3])
vector_v = np.array([4, 5, 6])

# compute the magnitude of the vector u
length_u = np.linalg.norm(vector_u)
print(f"{length_u:.3f}")

# make the vector a unit vector
vector_uu = vector_u / length_u
print(np.linalg.norm(vector_uu)) # output should be 1.0

# compute the dot product of the vectors
print(np.dot(vector_u, vector_v))

# compute the cross product of the vectors
print(np.cross(vector_u, vector_v))
```

```
3.742
1.0
32
[-3  6 -3]
```

```
# create two conformable matrices
# columns in matrix a = rows in matrix b
matrix_a = np.array([[1, 2, 3], [4, 5, 6]]) # 2 x 3 matrix
matrix_b = np.array([[7, 8], [9, 10], [11, 12]]) # 3 x 2 matrix

# multiply the matrices, this gives a 2 x 2 matrix
print(np.dot(matrix_a, matrix_b))
```

```
[[ 58  64]
 [139 154]]
```

```
# create a square (rows = columns) 3 x 3 matrix
matrix_c = np.array([[1, 7, 9], [3, 5, 8], [4, 2, 6]])

# compute the determinant of the matrix
print(np.around(np.linalg.det(matrix_c),4), "\n")

# compute the inverse of the matrix
matrix_ci = np.linalg.inv(matrix_c)
print(np.around(matrix_ci,4), "\n")

# the matrix times its inverse = the identity matrix
print(np.around(np.dot(matrix_c, matrix_ci),4))
```

```
-14.0

[[-1.      1.7143 -0.7857]
 [-1.      2.1429 -1.3571]
 [ 1.     -1.8571  1.1429]]

[[ 1.  0. -0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Here we use NumPy's linear algebra functions, some of which are found in the linalg module.

3.2 DataFrames

A DataFrame is a two-dimensional data structure in which information is organized in rows and columns, much like a table. To use DataFrames, we need to import the Pandas library[4]:

```
import os # import os library to work with directories
import pandas as pd # import pandas library under the alias pd
```

Let's see one example. Suppose we want to convert the following table into a DataFrame:

mineral	hardness	density
halite	2.5	2.17
quartz	7	2.65
hematite	6	5.3
rutile	6	4.24
olivine	7	3.4

We can do the following:

```
df = pd.DataFrame() # create an empty DataFrame

# add columns to the DataFrame using lists
# note that lists/columns should have the same length
df["mineral"] = ["halite", "quartz", "hematite",
                 "rutile", "olivine"]
df["hardness"] = [2.5, 7, 6, 6, 7]
df["density"] = [2.17, 2.65, 5.3, 4.24, 3.4]

df.head() # view first 5 rows of DataFrame
```

[4]The code for this section is included in the notebook `chapter3_2.ipynb`

	mineral	hardness	density
0	halite	2.5	2.17
1	quartz	7.0	2.65
2	hematite	6.0	5.30
3	rutile	6.0	4.24
4	olivine	7.0	3.40

Here, we added columns to the DataFrame using lists. The column on the far left—unlabeled by default—displays the index values, which help identify individual rows. Alternatively, we can create the DataFrame using a dictionary:

```
# create Dictionary
my_dict = {"mineral":["halite", "quartz", "hematite",
                      "rutile", "olivine"],
           "hardness":[2.5, 7, 6, 6, 7],
           "density":[2.17, 2.65, 5.3, 4.24, 3.4]}

# create DataFrame using Dictionary
df = pd.DataFrame(my_dict)

df.head() # view first 5 rows of DataFrame
```

	mineral	hardness	density
0	halite	2.5	2.17
1	quartz	7.0	2.65
2	hematite	6.0	5.30
3	rutile	6.0	4.24
4	olivine	7.0	3.40

In this case the dictionary keys are the names of the columns, and the values are the entries of the columns. What type of object is a DataFrame column? Let's see:

```
print(type(df["mineral"]),type(df["hardness"]),type(df["density"]))
```

```
<class 'pandas.core.series.Series'> <class 'pandas.core.series.Series'> <class '
    pandas.core.series.Series'>
```

Each column in a DataFrame is a Series. Unlike lists, Series offer greater flexibility and support vectorized operations. For example:

```
# add a column of hardness/density
df["hardness/density"] = df["hardness"] / df["density"]

df.head() # view first 5 rows of DataFrame
```

	mineral	hardness	density	hardness/density
0	halite	2.5	2.17	1.152074
1	quartz	7.0	2.65	2.641509
2	hematite	6.0	5.30	1.132075
3	rutile	6.0	4.24	1.415094
4	olivine	7.0	3.40	2.058824

3.2.1 NCS production data

To better illustrate the power of DataFrames, let's explore a practical example using monthly production data from oil and gas fields on the Norwegian Continental Shelf (NCS). This dataset is published by the Norwegian Offshore Directorate (NOD) and can be accessed here.

Let's read the data into a DataFrame:

```
# read field_production_monthly.csv file
# which is in the data directory

# path to file
path = os.path.join("..", "data", "field_production_monthly.csv")

# read csv file into DataFrame
df = pd.read_csv(path, sep=",")

df.head() # view first 5 rows of DataFrame
```

	prfInformationCarrier	prfYear	prfMonth	prfPrdOilNetMillSm3	prfPrdGasNetBillSm3	prfPrdNGLNet
0	16/1-12 Troldhaugen	2021	8	0.00653	0.00000	
1	16/1-12 Troldhaugen	2021	9	0.02573	0.00173	
2	16/1-12 Troldhaugen	2021	10	0.01172	0.00250	
3	16/1-12 Troldhaugen	2021	11	0.01298	0.00199	
4	16/1-12 Troldhaugen	2021	12	0.00461	0.00104	

In the cell above, we use Python's os library to construct a path to the CSV file[5]. We then load the data using Pandas' `read_csv()` method, which reads

[5] `os.path.join` automatically handles file path separators across different operating systems like Windows (\) and Unix/macOS (/)

comma-separated values into a DataFrame. Finally, we display the first five rows of the DataFrame using its `head()` method.

We can get more information about the DataFrame using its `info()` method:

```
df.info() # view DataFrame information
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26580 entries, 0 to 26579
Data columns (total 10 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   prfInformationCarrier               26580 non-null  object
 1   prfYear                             26580 non-null  int64
 2   prfMonth                            26580 non-null  int64
 3   prfPrdOilNetMillSm3                 26580 non-null  float64
 4   prfPrdGasNetBillSm3                 26580 non-null  float64
 5   prfPrdNGLNetMillSm3                 26580 non-null  float64
 6   prfPrdCondensateNetMillSm3          26580 non-null  float64
 7   prfPrdOeNetMillSm3                  26580 non-null  float64
 8   prfPrdProducedWaterInFieldMillSm3   26580 non-null  float64
 9   prfNpdidInformationCarrier          26580 non-null  int64
dtypes: float64(6), int64(3), object(1)
memory usage: 2.0+ MB
```

The DataFrame has 10 columns and 26,580 entries. The column names are rather long, so for convenience let's extract them into a list. We can use the DataFrame `columns` attribute to do that:

```
# column names
columns = df.columns.tolist()
print(columns)
```

```
['prfInformationCarrier', 'prfYear', 'prfMonth', 'prfPrdOilNetMillSm3', '
    prfPrdGasNetBillSm3', 'prfPrdNGLNetMillSm3', 'prfPrdCondensateNetMillSm3', '
    prfPrdOeNetMillSm3', 'prfPrdProducedWaterInFieldMillSm3', '
    prfNpdidInformationCarrier']
```

Let's filter the DataFrame to include only the fields operated by ConocoPhillips. First, we create a list of the relevant field names. Then, we filter the DataFrame by selecting rows where the first column (columns[0], which contains the field names) matches an entry in our list. This is done using the `isin()` method of a pandas Series. This method returns a Boolean Series, which we use to filter the DataFrame.

```
# list of ConocoPhillips fields
fields = ["EKOFISK", "ELDFISK", "TOMMELITEN GAMMA",
          "TOR", "VEST EKOFISK", "ALBUSKJELL", "VALHALL",
```

```
          "HOD", "TOMMELITEN A"]

# filter DataFrame to include only ConocoPhillips fields
df = df[df[columns[0]].isin(fields)]

df.head() # view first 5 rows of DataFrame
```

	prfInformationCarrier	prfYear	prfMonth	prfPrdOilNetMillSm3	prfPrdGasNetBillSm3	prfPrdNGLN
101	ALBUSKJELL	1979	5	0.00417	0.00768	
102	ALBUSKJELL	1979	6	0.01164	0.02090	
103	ALBUSKJELL	1979	7	0.03491	0.05398	
104	ALBUSKJELL	1979	8	0.09623	0.12506	
105	ALBUSKJELL	1979	9	0.11020	0.13332	

We can get the info for the filtered DataFrame:

```
df.info() # view DataFrame information
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3233 entries, 101 to 24545
Data columns (total 10 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   prfInformationCarrier               3233 non-null   object
 1   prfYear                             3233 non-null   int64
 2   prfMonth                            3233 non-null   int64
 3   prfPrdOilNetMillSm3                 3233 non-null   float64
 4   prfPrdGasNetBillSm3                 3233 non-null   float64
 5   prfPrdNGLNetMillSm3                 3233 non-null   float64
 6   prfPrdCondensateNetMillSm3          3233 non-null   float64
 7   prfPrdOeNetMillSm3                  3233 non-null   float64
 8   prfPrdProducedWaterInFieldMillSm3   3233 non-null   float64
 9   prfNpdidInformationCarrier          3233 non-null   int64
dtypes: float64(6), int64(3), object(1)
memory usage: 277.8+ KB
```

The DataFrame has now only 3,233 entries. Let's look at the production of Ekofisk. In the cell below, we make a smaller DataFrame for Ekofisk, and use the DataFrame `describe()` method to get a descriptive statistics of the DataFrame:

```
# let's look at the Ekofisk field
df_ekofisk = df[df[columns[0]] == "EKOFISK"]

df_ekofisk.describe() # describe DataFrame
```

	prfYear	prfMonth	prfPrdOilNetMillSm3	prfPrdGasNetBillSm3	prfPrdNGLNetMillSm3	p
count	644.000000	644.000000	644.000000	644.000000	644.000000	
mean	1997.748447	6.518634	0.783406	0.234457	0.044695	
std	15.506746	3.456186	0.406172	0.186144	0.034282	
min	1971.000000	1.000000	0.000190	0.000000	0.000000	
25%	1984.000000	4.000000	0.488335	0.074252	0.015675	
50%	1998.000000	7.000000	0.698255	0.209870	0.043155	
75%	2011.000000	10.000000	1.120433	0.394415	0.073847	
max	2025.000000	12.000000	1.719420	0.725610	0.114770	

The `describe()` method is quite powerful, it gives us the number of entries (644), the mean and standard deviation (std), minimum and maximum values (min and max), the lower and upper quartiles (25% and 75%), and the median (50%) of the column's values.

To end this example, let's calculate the oil and water production of Ekofisk in a given year. To do that, we filter the DataFrame to the year of interest, and sum the entries of the oil production (`columns[3]`) and water production (`columns[8]`) columns, using the Series `sum()` method:

```
year = 2015 # year of interest

# find out the oil and water production of EKOFISK in year
df_ekofisk = df_ekofisk[df_ekofisk[columns[1]] == year]

# sum montly oil production
oil_prod = df_ekofisk[columns[3]].sum()

# sum montly water production
water_prod = df_ekofisk[columns[8]].sum()

print(f"Ekofisk oil prod. in {year}: {oil_prod:.4f} Mill Sm3")
print(f"Ekofisk water prod. in {year}: {water_prod:.4f} Mill Sm3")
```

```
Ekofisk oil prod. in 2015: 6.4795 Mill Sm3
Ekofisk water prod. in 2015: 12.4423 Mill Sm3
```

I hope this example has given you a glimpse of the power of the Pandas library. We'll continue using Pandas throughout the course.

3.3 Exercises

1. The file `xeek_train_subset.csv` in the data folder, contains the logs of 12 wells from the Force 2020 Machine Learning lithology competition[6].

[6] This is a subset of the original dataset which is available at Andy McDonald's Petrophysics Python Series

(a) Load the file using the Pandas `read_csv()` method. Use the DataFrame `head()` and `info()` methods to learn more about the DataFrame.

(b) Extract the well `16/10-1` from the DataFrame. Hint: The well names are in column `WELL`.

(c) The column `FORCE_2020_LITHOFACIES_LITHOLOGY` contains lithology numbers. The significance of these numbers is as follows:

```
30000: Sandstone, 65030: Sandstone/Shale, 65000: Shale,
80000: Marl, 4000: Dolomite, 70000: Limestone,
70032: Chalk, 88000: Halite, 86000: Anhydrite,
99000: Tuff, 90000: Coal, 93000: Basement
```

Using a dictionary (keys are lithology numbers, values are lithology labels), create a new column called `LITH` with the lithology labels. Hint: Use the Series `map()` method and pass to this method the dictionary.

(d) The DataFrame `groupby()` function is a powerful tool used to split a DataFrame into groups based on one or more columns. Group the well by lithology (column `LITH`).

(e) Use the DataFrame `describe()` method to get a summary of the statistics of the different lithologies in the well.

2. This exercise builds on Exercise 1 and continues working with the well 16/10-1.

(a) Extract the Gamma Ray log of the well (column `GR`) to a NumPy array. Hint: You can use either the Series `values` attribute or the `to_numpy()` method.

(b) Smooth the GR curve with a 5-point moving average. Hint: Use the NumPy `convolve()` method. Pass to this method the GR array and the moving average filter.

(c) Define a threshold (e.g., GR $>$ 100 API) to identify shale-rich intervals.

(d) Extract the depths (column `DEPTH_MD`) that are likely shale.

Chapter 4

Control flow

Two fundamental concepts form the backbone of computer programs: decision-making (yes or no) and repetition. By default, Python executes code sequentially—one line at a time. However, real-world problems often require more flexible control over that flow. Sometimes, we need to check a condition and, depending on the result, execute certain parts of the code either once or multiple times. To make this possible, Python provides tools for conditional and iterative execution.

4.1 Conditional execution

For conditional execution, Python provides the `if-else` statement. The syntax of this statement is as follows:

```
if␣expression:
␣␣␣␣statement_1
else:
␣␣␣␣statement_2
```

Python evaluates the expression inside the `if` statement. If the condition is true, then `statement_1` is executed; otherwise, `statement_2` runs. Notice that both `if` and `else` statements end with a colon (`:`). This signals the start of a block of code. The code inside these blocks must be indented, typically by four spaces (␣). In Python, indentation isn't just for readability—it's how the language defines which statements belong to which block. Figure 1 below illustrates how conditional execution works.

Here are few examples[1]:

[1]The code for this chapter is included in the notebook `chapter4.ipynb`

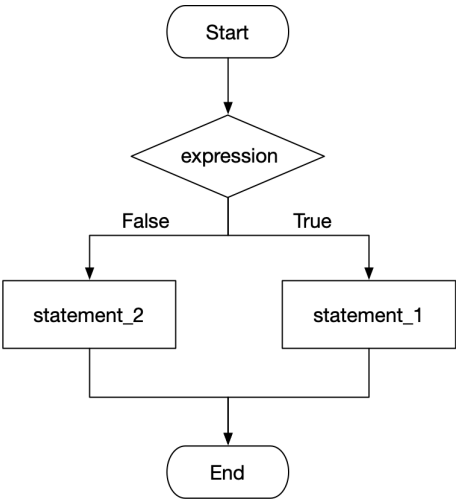


Figure 4.1: Conditional execution.

```
# import libraries required for the notebook
import random # import random library
import requests # import requests (for HTTP requests)
from io import StringIO # import StringIO (for reading the data)
import numpy as np # import numpy as np
import pandas as pd # import pandas as pd
```

```
# convert from Celsius to Fahrenheit or vice versa

# define temperature unit and value
t_type = "C"
t_val = 23.0

# do the conversion
if t_type == "C":
    tc_type = "F"
    tc_val = 9 / 5 * t_val + 32
else:
    tc_type = "C"
    tc_val = 5 / 9 * (t_val - 32)

print(f"{t_val:.2f} {t_type} = {tc_val:.2f} {tc_type}")
```

```
23.00 C = 73.40 F
```

```
# the else statement is sometimes not needed

my_vector = np.array([1, 2, 3])
```

```
# if my_vector is not a unit vector make it a unit vector
if np.linalg.norm(my_vector) != 1.0:
    my_vector = my_vector / np.linalg.norm(my_vector)

print(f"vector magnitude = {np.linalg.norm(my_vector):.2f}")
```

```
vector magnitude = 1.00
```

```
# it is possible to include more than one
# condition using the elif statement

azimuth = 232 # angle between 0 and 360 deg

if azimuth == 0 or azimuth == 360:
    direction = "N"
elif azimuth > 0 and azimuth < 90:
    direction = "NE"
elif azimuth == 90:
    direction = "E"
elif azimuth > 90 and azimuth < 180:
    direction = "SE"
elif azimuth == 180:
    direction = "S"
elif azimuth > 180 and azimuth < 270:
    direction = "SW"
elif azimuth == 270:
    direction = "W"
else:
    direction = "NW"

print(f"Azimuth {azimuth} = {direction} direction")
```

```
Azimuth 232 = SW direction
```

In the cell above, `and` and `or` are logical operators. They are used to combine conditional statements. Another logical operator is `not`, which negates a Boolean value-returning `False` if the condition is `True`, and vice versa. You can learn more about logical operators here.

Let's move on to something more interesting—reading monthly production data for fields on the NCS directly from Factpages. The code below handles this task by first constructing the appropriate URL[2]. It then uses Python's requests library to fetch the data. If the request is successful, the data is read into a DataFrame; otherwise, an error message is displayed[3].

[2]Last checked May, 2025

[3]Depending on your organization's security settings, you may need to verify the SSL certificate when using requests.get(), by providing a custom certificate with `verify="path_to_certificate"`

```
# read data from NOD factpages

# construct the URL
u_1 = "https://factpages.sodir.no/public?/Factpages/external/tableview/"
u_2 = "&rs:Command=Render&rc:Toolbar=false&rc:Parameters=f"
u_3 = "&IpAddress=not_used&CultureCode=en&rs:Format=CSV&Top100=false"
descriptor = "field_production_monthly"
url = u_1 + descriptor + u_2 + u_3

# request the data
response = requests.get(url)
# if the request was successful
if response.status_code == 200:
    # load csv data into a DataFrame. StringIO wraps the string
    # so it can be read as a file
    df = pd.read_csv(StringIO(response.text))
    # output DataFrame info
    df.info()
else:
    print(f"Error: {response.status_code}")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26580 entries, 0 to 26579
Data columns (total 10 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   prfInformationCarrier               26580 non-null  object
 1   prfYear                             26580 non-null  int64
 2   prfMonth                            26580 non-null  int64
 3   prfPrdOilNetMillSm3                 26580 non-null  float64
 4   prfPrdGasNetBillSm3                 26580 non-null  float64
 5   prfPrdNGLNetMillSm3                 26580 non-null  float64
 6   prfPrdCondensateNetMillSm3          26580 non-null  float64
 7   prfPrdOeNetMillSm3                  26580 non-null  float64
 8   prfPrdProducedWaterInFieldMillSm3   26580 non-null  float64
 9   prfNpdidInformationCarrier          26580 non-null  int64
dtypes: float64(6), int64(3), object(1)
memory usage: 2.0+ MB
```

4.2 Iterative execution

Python offers two types of loops for iterative execution: `while` loops and `for` loops. These are used when you need to run a block of code multiple times.

4.2.1 while loop

The syntax of the `while` loop is as follows:

```
initialize_variable

while␣expression:
```

```
␣␣␣␣statement
␣␣␣␣modify_variable
```

We start by initializing a variable. Then, Python evaluates an expression that involves this variable. If the expression is true, the loop's body is executed; if not, the program exits the loop. The loop continues to run as long as the expression remains true. It is important to update the variable inside the loop—otherwise, the condition may never become false, resulting in an infinite loop. Figure 4.2a illustrates this process.

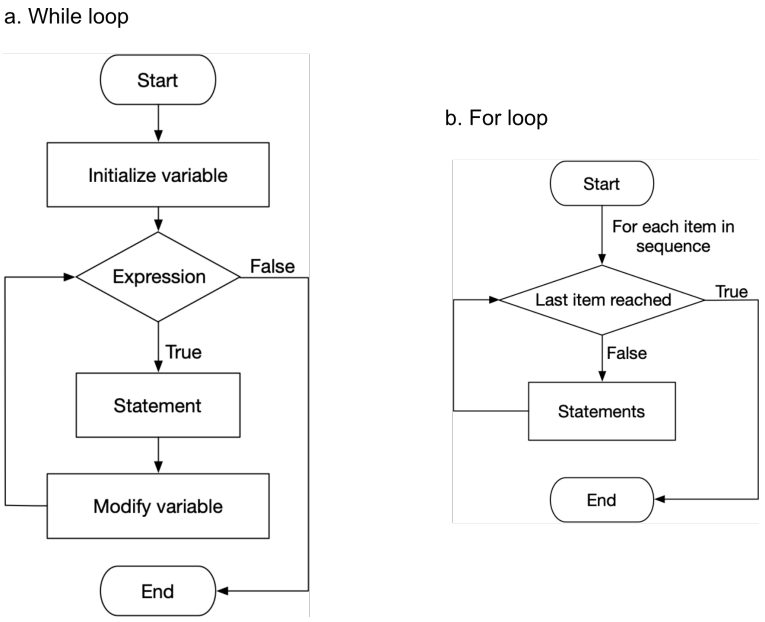


Figure 4.2: **a.** while and **b.** for loops.

These are some examples:

```
# sum numbers from 0 to 99_999 that are multiple of 3 or 5

number = 0 # number
sum = 0 # sum
while number < 100_000:
    # if number is multiple of 3 or 5
    if number % 3 == 0 or number % 5 == 0:
        sum += number # add number to sum
    number += 1 # increment number

print(f"Sum of numbers = {sum}")
```

```
Sum of numbers = 2333316668
```

```
# calculate pressure in a column of
# 200 m water and 300 m rock

g = 9.81 # gravity in m/s^2
rho_w = 1000 # water density in kg/m^3
d_w = 200 # water depth in m
p_w = rho_w * g * d_w # pressure at sea bottom in Pa
rho_r = 2700 # rock density in kg/m^3

d = 0.0 # initial depth in m

# table heading: <10 means left aligned with 10 characters
print(f"{"Depth[m]":<10} Pressure[kPa]")
print("-" * 25)

while d <= 500.0: # while depth is < 500 m
    if d <= d_w: # if in water
        p = rho_w * g * d
    else: # else if in rock
        p = p_w + rho_r * g * (d - d_w)
    # print depth and pressure in kPa
    print(f"{d:<10.1f} {p*1e-3:.1f}")
    # update depth
    d += 50.0
```

```
Depth[m]   Pressure[kPa]
-------------------------
0.0        0.0
50.0       490.5
100.0      981.0
150.0      1471.5
200.0      1962.0
250.0      3286.3
300.0      4610.7
350.0      5935.1
400.0      7259.4
450.0      8583.8
500.0      9908.1
```

The `while` loop is especially useful in situations where the number of iterations isn't known in advance. For example:

```
# simulate rolling two dice until double sixes are rolled

attempts = 0
die1 = die2 = 0

while die1 != 6 or die2 != 6:
    die1 = random.randint(1, 6)
    die2 = random.randint(1, 6)
```

```
    attempts += 1

print(f"It took {attempts} roll(s) to get double sixes!")
```

```
It took 30 roll(s) to get double sixes!
```

We don't know how many attempts it will take to roll double sixes — it's entirely random. This example also demonstrates the use of Python's built-in random module.

4.2.2 for loop

The `for` loop has a simpler syntax:

```
for item in sequence:
    statement
```

The syntax comprises an `item` and a `sequence`. The sequence can be any collection of data. During the execution of the loop, the first element of the sequence is assigned to `item` and the statement(s) of the loop body are executed, then the next element is assigned to `item` and the statement(s) are again executed, and so on until all elements of the sequence are exhausted. Figure 4.2b illustrates this process.

Let's try the examples above, but this time using a `for` loop:

```
# sum numbers from 0 to 99_999 that are multiple of 3 or 5

sum = 0 # sum

for number in np.arange(100_000): # for number 0 to 99_999
    # if number is multiple of 3 or 5
    if number % 3 == 0 or number % 5 == 0:
        sum += number # add number to sum

print(f"Sum of numbers = {sum}")
```

```
Sum of numbers = 2333316668
```

```
# calculate pressure in a column of
# 200 m water and 300 m rock

# input variables as before

# table heading
print(f"{"Depth[m]":<10} Pressure[kPa]")
print("-" * 25)
```

```
# depths from 0 to 500 m
ds = np.arange(0, 501, 50)
#ds = np.linspace(0, 500, 11) # another option

for d in ds:
    if d <= d_w: # if in water
        p = rho_w * g * d
    else: # else if in rock
        p = p_w + rho_r * g * (d - d_w)
    # print depth and pressure in kPa
    print(f"{d:<10.1f} {p*1e-3:.1f}")
```

```
Depth[m]   Pressure[kPa]
-------------------------
0.0        0.0
50.0       490.5
100.0      981.0
150.0      1471.5
200.0      1962.0
250.0      3286.3
300.0      4610.7
350.0      5935.1
400.0      7259.4
450.0      8583.8
500.0      9908.1
```

In the two examples above, we use NumPy's `arange()` function to create an array that starts at a given value (default is 0) and goes up to, but does not include, the end value, using a specified step size (default is 1). Alternatively, the `linspace()` function can be used to create an array from start to end, but instead of defining the step size, you specify the total number of elements. In short, `arange()` lets you control the step size, while `linspace()` lets you control the number of steps.

4.2.3 zip, enumerate and break

In `for` loops, it's often useful to iterate over more than one sequence (like lists) at the same time—this is where the built-in `zip()` function comes in handy, by pairing elements together from each iterable. Here is one example:

```
minerals = ["Quartz", "Gypsum", "Talc"]
hardness = [7, 2, 1]

for mineral, hard in zip(minerals, hardness):
    print(f"{mineral} hardness is {hard}")
```

```
Quartz hardness is 7
Gypsum hardness is 2
Talc hardness is 1
```

It is also possible to extract the index of the iteration along with the elements by using the built-in `enumerate()` function, which makes it easy to track the position in the sequence while iterating:

```
for i, (mineral, hard) in enumerate(zip(minerals, hardness)):
    print(f"{i+1}. {mineral} hardness is {hard}")
```

```
1. Quartz hardness is 7
2. Gypsum hardness is 2
3. Talc hardness is 1
```

Finally, you can use the `break` statement to stop a loop and exit it. Here's an example:

```
number = random.randint(0, 100) # random number between 0 and 100

while True:
    # ask user for a guess, int is used to convert input to int
    guess = int(input("Guess the number: "))
    if guess == number: # if guess is correct
        print("You guessed it!")
        break # exit the loop
    elif guess < number: # if guess is too low
        print("Too low!")
    else: # if guess is too high
        print("Too high!")
```

```
Too low!
Too low!
Too high!
You guessed it!
```

4.3 Vectorization

Vectorized operations can significantly improve the performance of your code. Below is a vectorized version of the depth versus pressure code above. It gives the same output:

```
# table heading
print(f"{"Depth[m]":<10} Pressure[kPa]")
print("-" * 25)

water = ds <= 200.0 # boolean array for water
rock = ds > 200.0  # boolean array for rock

# calculate pressures, this gives an array
ps = rho_w * g * ds * water + (p_w + rho_r * g * (ds - d_w)) * rock
```

```
for d, p in zip(ds, ps): # iterate over depth and pressure
    # print depth and pressure in kPa
    print(f"{d:<10.1f} {p*1e-3:.1f}")
```

The boolean arrays `water` and `rock` indicate the depths at which these materials are present. Pressures (`ps`) are calculated in a single line using vectorized operations. In this calculation, the `water` and `rock` arrays are automatically converted by Python: `True` values become `1` and `False` values become `0`. Finally, the `for` loop outputs the depths and corresponding pressures. The `zip()` function allows simultaneous iteration over the `ds` (depths) and `ps` (pressures) arrays.

While the performance gain in this example is minimal—since the code involves few operations—the advantages of vectorization become much more apparent in computation-heavy tasks. When working with large datasets or complex numerical calculations, it's a good practice to consider vectorization. However, aim for a balance between performance and clarity to ensure your code remains efficient and easy to understand (Figure 4.3).

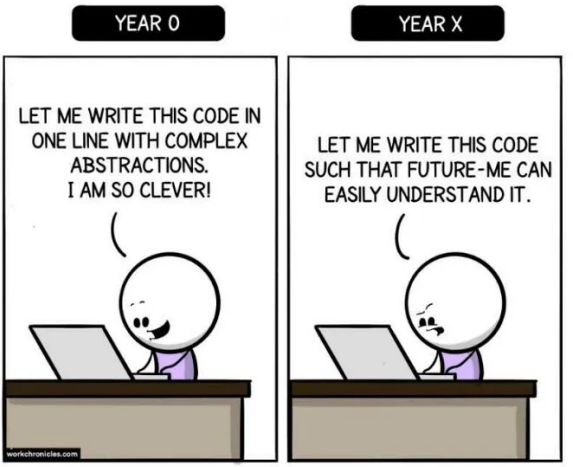


Figure 4.3: Balance between performance and clarity. From Work Chronicles.

4.4 List comprehensions

List comprehensions are a concise and powerful feature in Python that allow you to quickly create new lists by looping over a collection and optionally filtering its elements. The basic syntax is:

```
[expression for value in collection if condition]
```

List comprehensions combine looping and conditionals into a single, readable line. The conditional part is optional. Let's revisit the same two examples:

```
# sum numbers from 0 to 99_999 that are multiple of 3 or 5

numbers = np.arange(100_000) # numbers

# find multiples using list comprehension
multiples = [number for number in numbers
             if number % 3 == 0 or number % 5 == 0]

sum = np.sum(multiples) # sum multiples

print(f"Sum of numbers = {sum}")
```

```
Sum of numbers = 2333316668
```

```
# table heading
print(f"{"Depth[m]":<10} Pressure[kPa]")
print("-" * 25)

# calculate pressures using list comprehension
ps = [rho_w*g*d if d <= 200.0 else p_w+rho_r*g*(d-d_w) for d in ds]

for d, p in zip(ds, ps):
    # print depth and pressure in kPa
    print(f"{d:<10.1f} {p*1e-3:.1f}")
```

This is pretty slick but perhaps not as clear as using a loop to calculate pressures (Figure 4.3).

4.5 Exercises

1. The file `xeek_train_subset.csv` contains well log data including gamma ray (`GR`) values and stratigraphic information.

   (a) Extract the data for well 16/10-1 into a Pandas DataFrame.

   (b) From the gamma ray log (column `GR`), compute a new column called `VSH` (Volume of Shale) using different equations depending on the age of the rocks. For rocks older than the Tertiary, use the following equation:

$$\mathrm{VSH} = \mathrm{IGR} = (\mathrm{GR} - \mathrm{GR\,min})/(\mathrm{GR\,max} - \mathrm{GR\,min})$$

where GR is the gamma ray value for the sample, GR min is the minimum GR (0), and GR max is the maximum GR (200).

For Tertiary and younger rocks, specifically the Nordland, Hordaland and Rogaland groups, use:

$$\text{VSH} = 0.083 * (2^{(3.7* \text{IGR})} - 1)$$

Hints:

- You should first compute `IGR` using the formula for all rows.
- Then apply the appropriate VSH equation depending on the group name (in the `GROUP` column).
- Add the resulting `VSH` values as a new column in the DataFrame.

2. You are provided with two datasets:

   - `group_colors.csv`: Contains color information for each geological group. Each color is represented as a list of three integers corresponding to red, green, and blue (RGB) values, ranging from 0 to 255.
   - `xeek_train_subset.csv`: Contains log data for multiple wells, including a column named GROUP indicating the group for each record.

   For the well 16/10-1, create a dictionary where:

   - Each key is the name of a group found in that well.
   - Each value is a list representing the RGB color of the group, normalized so that each component is between 0 and 1 (by dividing by 255), and rounded to two decimal places.

3. This exercise builds on Exercise 2 and continues working with the well 16/10-1.

   (a) Create a dictionary that maps each geological group present in the well to its top depth defined as the shallowest `DEPTH_MD` value where that group appears.

   Hints:

   - Filter the data to include only rows for well 16/10-1.
   - For each unique group in the `GROUP` column, find the minimum value in the `DEPTH_MD` column where that group occurs.
   - Construct a dictionary where the keys are the group names, and the values are the corresponding top depths.

(b) The above procedure will not work if the interval of interest is repeated in the well. For groups this does not seem to be a problem, but for facies (column `FORCE_2020_LITHOFACIES_LITHOLOGY`) it is. Create a dictionary that maps each facies in the well to its top depth.

Hints:

- Find the top depth of each facies. This is the first row where a new facies appears.
- Build a dictionary mapping facies to their top depths. If a facies appears more than once, append the row index to the key (e.g. "65000-3").

Chapter 5

Organizing code

Getting code to run is only the beginning. As your projects grow, keeping things organized becomes just as important as making them work. Without structure, your code can quickly turn into spaghetti code—a tangled mess that's hard to understand, reuse, or build on.

Good organization is key to writing code that lasts. Whether you're working solo or with others, how you structure your code affects how easily it can be read, maintained, and improved.

In this chapter, we'll explore how functions, classes, and modular programming bring clarity and structure to your code, helping you build programs that are both robust and scalable.

5.1 Functions

A function is a reusable block of code that performs a specific task. It allows you to group related instructions under a single name, so you can run them whenever you need—without repeating yourself. Functions help make your code more organized, readable, and easier to maintain.

The syntax of a function is as follows:

```
def function_name(arg_1, arg_2...arg_n):
    statements
    return val
```

`def` is the header of the function, it generates the function object and assigns a name to it. In the parentheses, the input parameters are included. When the

function does not have any input parameter, then the parentheses is left empty. After the colon (`:`), the function statements are included. The return statement returns a value. If `val` is not specified, the function returns `None`. Both `val` and the `return` statement are optional.

Let's turn the code that reads data from Factpages into a function[1]:

```
# import libraries required for the notebook
import os # import os to work with directories
import requests # import requests (for HTTP requests)
from io import StringIO # import StringIO (for reading the data)
import numpy as np # import numpy as np
import pandas as pd # import pandas as pd
import matplotlib.pyplot as plt # import matplotlib.pyplot as plt
from scipy import integrate # import scipy integrate module
```

```
def read_factpages(descriptor):
    """
    Read data from NOD factpages
    Input:
        descriptor: String with NOD descriptor,
        e.g. "field_production_monthly"
    Output:
        df: DataFrame with the data or
        empty if data could not be read
    """
    # construct the URL
    u_1 = "https://factpages.sodir.no/public?/Factpages/external/tableview/"
    u_2 = "&rs:Command=Render&rc:Toolbar=false&rc:Parameters=f"
    u_3 = "&IpAddress=not_used&CultureCode=en&rs:Format=CSV&Top100=false"
    url = u_1 + descriptor + u_2 + u_3

    # create an empty DataFrame
    df = pd.DataFrame()

    # request the data
    response = requests.get(url)
    # if the request was successful
    if response.status_code == 200:
        # load csv data into a DataFrame
        df = pd.read_csv(StringIO(response.text))
    else:
        print(f"Error: {response.status_code}")

    return df
```

The text enclosed in triple quotes is called a *docstring*—it describes what the function does. Including a clear and concise docstring is always a good habit, as it helps others (and your future self) understand the purpose of the function.

[1]The code for this chapter is included in the notebook `chapter5.ipynb`

In VS Code, this information can be accessed anytime using IntelliSense (see Figure 5.1).

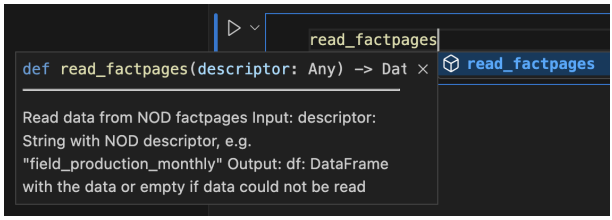


Figure 5.1: IntelliSense on our function.

Now we can use the function to read data:

```
# monthly field production
df = read_factpages("field_production_monthly")
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26580 entries, 0 to 26579
Data columns (total 10 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
 0   prfInformationCarrier             26580 non-null  object
 1   prfYear                           26580 non-null  int64
 2   prfMonth                          26580 non-null  int64
 3   prfPrdOilNetMillSm3               26580 non-null  float64
 4   prfPrdGasNetBillSm3               26580 non-null  float64
 5   prfPrdNGLNetMillSm3               26580 non-null  float64
 6   prfPrdCondensateNetMillSm3        26580 non-null  float64
 7   prfPrdOeNetMillSm3                26580 non-null  float64
 8   prfPrdProducedWaterInFieldMillSm3 26580 non-null  float64
 9   prfNpdidInformationCarrier        26580 non-null  int64
dtypes: float64(6), int64(3), object(1)
memory usage: 2.0+ MB
```

By accepting a `descriptor` as input, the function becomes flexible and reusable—it can read any dataset from Factpages, as long as we know the corresponding `descriptor`. For example:

```
# long list of all exploration wells
df = read_factpages("wellbore_exploration_all")
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2146 entries, 0 to 2145
Data columns (total 87 columns):
```

```
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   wlbWellboreName             2146 non-null   object
 1   wlbWell                     2146 non-null   object
 2   wlbDrillingOperator         2146 non-null   object
 3   wlbProductionLicence        2142 non-null   object
 4   wlbPurpose                  2145 non-null   object
 5   wlbStatus                   2136 non-null   object
 6   wlbContent                  2083 non-null   object
 7   wlbWellType                 2146 non-null   object
...
 85  wlbDateUpdatedMax           2016 non-null   object
 86  datesyncNPD                 2146 non-null   object
dtypes: float64(19), int64(14), object(54)
memory usage: 1.4+ MB
Output is truncated. View as a scrollable element or open in a text editor.
     Adjust cell output settings...
```

Let's try another example. The area of a polygon of any shape (except one that crosses itself) can be written as:

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1}y_i) \text{ if } i+1 > n, i+1 = 1 \tag{5.1}$$

where n is the number of points in the polygon, and (x_i, y_i) are the x and y coordinates of the points. For example, for a four points polygon the area is:

$$A = \frac{1}{2}(x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4) \tag{5.2}$$

This equation returns a positive area if the points are ordered counter-clockwise, or a negative area if the points are ordered clockwise. Let's implement a function that computes the area of a polygon using this formula:

```
def polyg_area(x,y):
    """
    calculate and return the area of a polygon
    from the x and y coordinates of its points
    Note: points must be in sequential order
    """
    npoints = x.size # number of points
    area = 0.0 # initialize area

    # calculate polygon's area
    for i in range(npoints):
        # index of the next point
        next_i = i + 1
        if i == npoints-1: # if last point
            next_i = 0
        # calculate area
        area += (x[i]*y[next_i] - y[i]*x[next_i])
```

```
    # return area
    return np.abs(area)/2
```

Surprisingly, this function can be shortened to one line of code![2]

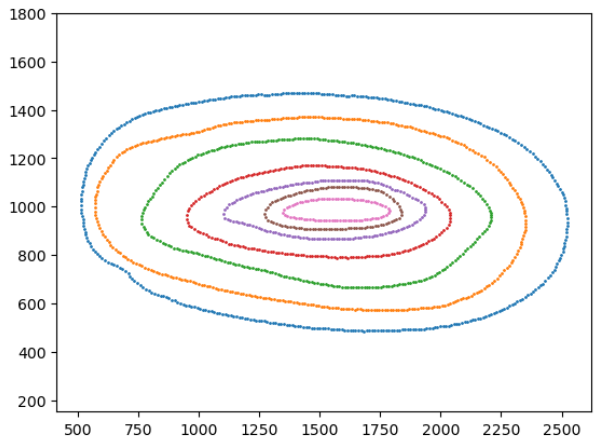
```
def polyg_area(x,y):
    """
    calculate and return the area of a polygon
    from the x and y coordinates of its points
    Note: points must be in sequential order
    """
    return 0.5*np.abs(np.dot(x, np.roll(y,1))
                      - np.dot(y, np.roll(x,1)))
```

The file `net_oil.txt` in the data directory, contains the `x`(east), `y` (north), and `z` (value) of contours in an isochore map (vertical thickness) of net oil in a trap. All values are in meters. Let's compute the volume of the trap. We first determine the area of the contours using our function:

```
path = os.path.join("..", "data", "net_oil.txt")
contours = np.loadtxt(path) # read the contours

c_values = np.unique(contours[:,2]) # contour values
c_areas = np.zeros(c_values.size) # initialize areas

for i in range(c_values.size):
    # extract contour
    contour = contours[contours[:,2] == c_values[i]]
    # calculate area
    c_areas[i] = polyg_area(contour[:,0],contour[:,1])
    # plot contour
    plt.plot(contour[:,0],contour[:,1],".",markersize=2)

print("Areas in m2 = ", c_areas.astype(int)) # print areas
plt.axis("equal") # set equal axis
plt.show() # show plot
```

[2]To understand why this line works, check the NumPy roll and dot methods.

```
Areas in m2 =  [1602030 1126504  659259  315659  150486   78587   31450]
```



In the code above, we also plotted the contours using the Matplotlib library. While this will be covered in more detail in Chapter 6, for now just notice how straightforward it is to visualize data in Python.

Mathematically, the volume of the trap can be expressed as:

$$V = \int_a^b A(z)dz \tag{5.3}$$

To understand why this works, imagine slicing the volume into many horizontal layers—these are the contours. We can estimate the volume between each pair of adjacent contours and then sum them all to obtain the total volume.

So, to calculate the volume, we need to perform an integration. We can do this using the scipy.integrate module. In the code below, we use the trapezoidal rule—implemented as the `trapezoid()` method—to carry out the integration:

```
v_oil = integrate.trapezoid(c_areas, c_values) # volume of oil
print(f"Volume of oil: {v_oil:.0f} m3 or {v_oil*6.2898:.0f} bbl")
```

```
Volume of oil: 15736180 m3 or 98977425 bbl
```

5.2 Classes

Object-oriented design underpins many of Python's libraries and tools. Python is fundamentally an object-oriented programming (OOP) language. It organizes code around objects—structures that combine data with the functions (called

methods) that operate on that data. The two main building blocks of OOP are classes and objects.

A class is a blueprint for creating objects. It defines the shared structure and behavior its instances will have. For example, an `Employee` class might include attributes like `name` and `salary`, and a method like `raise_salary`. From this class, you can create multiple employee objects, each with their own data but the same general behavior.

Python makes working with classes simple. Here's a basic class definition:

```
class ClassName:
    def __init__(self, parameters):
        # initialization code
        self.attribute = value

    def method(self):
        # method code
```

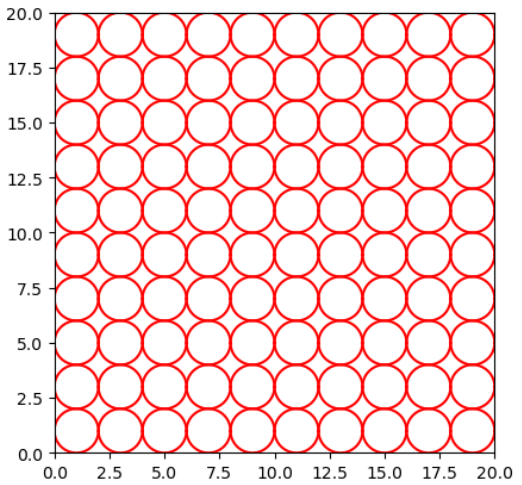
Let's look at a simple example that defines a `Circle` class:

```
class Circle:
    """
    A class that implements a circle
    """
    # initialization requires center [x, y]
    # and radius of circle
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    # methods

    # circumference
    def circumference(self):
        return 2 * np.pi * self.radius

    # area
    def area(self):
        return np.pi * self.radius ** 2

    # x and y coordinates defining circle
    def coordinates(self):
        theta = np.arange(0,360) * np.pi / 180
        x = self.radius * np.cos(theta) + self.center[0]
        y = self.radius * np.sin(theta) + self.center[1]
        return x, y

    # shift center in x
    def shift_in_x(self, x_value):
        self.center[0] += x_value
```

```
    # shift center in y
    def shift_in_y(self, y_value):
        self.center[1] += y_value
```

Now let's use this class to fill a 20×20 unit square with circles of radius 1. We'll also calculate the areal porosity, which measures the fraction of the area not occupied by the circles.

```
area_circles = 0.0 # initialize circles area
my_circle = Circle([-1, -1],1) # unit circle with center -1,-1

# use two nested loops
# i moves circle in y
# j moves circle in x
for i in range(10):
    my_circle.center[0] = -1 # reset x of circle center to -1
    my_circle.shift_in_y(2) # shift circle in y 2 units
    for j in range(10):
        my_circle.shift_in_x(2) # shift circle in x 2 units
        area_circles += my_circle.area() # update circles area
        x, y = my_circle.coordinates() # circle coordinates
        plt.plot(x,y,'r-') # plot circle

plt.axis("square") # square axis
plt.xlim([0, 20]) # x axis limits
plt.ylim([0, 20]) # y axis limits

# estimate and print areal porosity
area_total = 20 * 20
area_voids = area_total - area_circles
print(f"Areal porosity = {area_voids/area_total:.2f}")
```




```
Areal porosity = 0.21
```

Note that only a single `Circle` instance (`my_circle`) is created in the second line of the code. This circle is then shifted in the `y` (`shift_in_y()`) and `x` (`shift_in_x()`) directions within two nested loops to fill the square. In each iteration, the circle area is calculated using the `area()` method and added to the total. The coordinates of points along the circumference are generated using the `coordinates()` method and plotted with the Matplotlib `plot()` function.

Let's look at another example. Building on our code to read data from Factpages, we'll now define a top-level class responsible for setting up the main URL components and loading data from a descriptor.

```
class FP_reader:
    """
    Class to read NOD factpages
    """
    def __init__(self):
        """
        initialize strings to construct
        the URL as of May 2025
        """
        self.u_1 = "https://factpages.sodir.no/public?/Factpages/external/tableview/"
        self.u_2 = "&rs:Command=Render&rc:Toolbar=false&rc:Parameters=f"
        self.u_3 = "&IpAddress=not_used&CultureCode=en&rs:Format=CSV&Top100=false"

    def read(self, descriptor):
        """
        Read data from NOD factpages
        Input:
            descriptor: NOD descriptor,
            e.g. "field_production_monthly"
        Output:
            df: DataFrame with the data or
            empty if data could not be read
        """
        # construct the URL
        url = self.u_1 + descriptor + self.u_2 + self.u_3

        # request the data
        response = requests.get(url)
        # if the request was successful
        if response.status_code == 200:
            # load csv data into a DataFrame
            self.df = pd.read_csv(StringIO(response.text))
        else:
            print(f"Error: {response.status_code}")

        return self.df
```

Next, we'll create a class that inherits from the top-level class and is responsible for reading data from specific fields.

```
class Field(FP_reader):
    """
    Class to read field data from NOD factpages
    """
    def __init__(self):
        """
```

```
        Initialize field class
        """
        # call parent class
        super().__init__()

    def monthly_production(self, fields=[]):
        """
        Read field production monthly data
        Input:
            fields: list of field names. Pass
            empty list to read all fields
        Output:
            df: DataFrame with the data or
            empty if data could not be read
        """
        # call parent class method
        df = self.read("field_production_monthly")
        # if df and fields are not empty
        if not df.empty and len(fields) > 0:
            # filter by field names
            df = df[df["prfInformationCarrier"].isin(fields)]

        return df
```

Note that the `Field` class currently has just one method, but it's easy to extend the class by adding more methods to read additional datasets from the *Field* category in Factpages. Let's read the monthly production from ConocoPhillips fields using the class:

```
# read monthly production data
# of ConocoPhillips fields
fields = ["EKOFISK", "ELDFISK", "TOMMELITEN GAMMA",
          "TOR", "VEST EKOFISK", "ALBUSKJELL", "VALHALL",
          "HOD", "TOMMELITEN A"]

field = Field() # create field object
df = field.monthly_production(fields) # read data
df.info() # print info
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3233 entries, 101 to 24545
Data columns (total 10 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   prfInformationCarrier               3233 non-null   object
 1   prfYear                             3233 non-null   int64
 2   prfMonth                            3233 non-null   int64
 3   prfPrdOilNetMillSm3                 3233 non-null   float64
 4   prfPrdGasNetBillSm3                 3233 non-null   float64
 5   prfPrdNGLNetMillSm3                 3233 non-null   float64
 6   prfPrdCondensateNetMillSm3          3233 non-null   float64
 7   prfPrdOeNetMillSm3                  3233 non-null   float64
 8   prfPrdProducedWaterInFieldMillSm3   3233 non-null   float64
```

```
 9   prfNpdidInformationCarrier         3233 non-null   int64
dtypes: float64(6), int64(3), object(1)
memory usage: 277.8+ KB
```

5.3 Modular programming

The functions and classes we've created so far are only available within the current notebook. Wouldn't it be convenient to store them in a way that allows us to easily import and use them in any other file when needed? In Python, we can achieve this by organizing them into *modules* and *packages*.

A module is a Python source file (`.py`) that contains code designed to perform a specific task. For example, the first Python file we created in this course, `my_first_code.py`, is actually a module. A package is a directory that contains modules or data, along with an initialization file (`__init__.py`), which signals to Python that the directory should be treated as a package.

To illustrate this, let's create a package called `factpages` to handle reading data from Factpages. In your `code` directory, create a folder called `factpages`. Within `factpages`, create the files `__init__.py`, `reader.py` and `field.py`. The `factpages` directory should look like Figure 5.2.

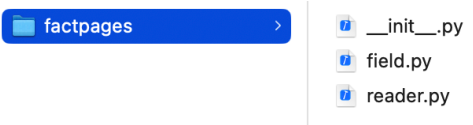


Figure 5.2: The factpages package with modules reader and field.

In the `reader.py` file, paste the code we created to define the `FP_reader` class [3]:

```python
import requests
import pandas as pd

class FP_reader:
    """
    Class to read NOD factpages
    """
    def __init__(self):...

    def read(self, descriptor):..
```

[3]The functions are collapsed for brevity. To view the full code, please refer to the corresponding file.

In the `field.py` file, paste the code we previously wrote to define the `Field` class. Additional functions for working with the *Field* category of Factpages are also included. Note that the `field` module imports the class `FP_reader` from the `reader` module. This is necessary since `Field` inherits from `FP_reader`.

```
from .reader import FP_reader # import class FP_reader

class Field(FP_reader):
    """
    Class to read field data from NOD factpages
    """
    def __init__(self):...

    def monthly_production(self, fields=[]):...

    def yearly_production(self, fields=[]):...

    def monthly_total_production(self):...

    def yearly_total_production(self):...

    def operators(self, fields=[]):...

    def reserves(self, fields=[]):...

    def inplace_volumes(self, fields=[]):...

    def investments(self, fields=[]):...

    def expected_investments(self, fields=[]):...

    def description(self, fields=[]):...
```

We want to be able to call the functions directly from the package. To achieve this, add the following code to the `__init__.py` file:

```
# load the modules in the package
from .reader import *
from .field import *
```

This will import all functions from the modules into the main package namespace, making them accessible directly from the package.

Now we can use our package. Before running the cell below, make sure to clear all outputs and restart the kernel. This ensures that we're starting with a clean slate.

```
import factpages as fp # import our package

# read monthly production data
```

```
# of ConocoPhillips fields
fields = ["EKOFISK", "ELDFISK", "TOMMELITEN GAMMA",
          "TOR", "VEST EKOFISK", "ALBUSKJELL", "VALHALL",
          "HOD", "TOMMELITEN A"]
field = fp.Field() # create field object
df = field.monthly_production(fields) # read monthly production data
df.info() # print info
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3233 entries, 101 to 24545
Data columns (total 10 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   prfInformationCarrier               3233 non-null   object
 1   prfYear                             3233 non-null   int64
 2   prfMonth                            3233 non-null   int64
 3   prfPrdOilNetMillSm3                 3233 non-null   float64
 4   prfPrdGasNetBillSm3                 3233 non-null   float64
 5   prfPrdNGLNetMillSm3                 3233 non-null   float64
 6   prfPrdCondensateNetMillSm3          3233 non-null   float64
 7   prfPrdOeNetMillSm3                  3233 non-null   float64
 8   prfPrdProducedWaterInFieldMillSm3   3233 non-null   float64
 9   prfNpdidInformationCarrier          3233 non-null   int64
dtypes: float64(6), int64(3), object(1)
memory usage: 277.8+ KB
```

And we can try another method:

```
df = field.investments(fields) # read investments data
df.info() # print info
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 427 entries, 0 to 3301
Data columns (total 5 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   prfInformationCarrier        427 non-null    object
 1   prfYear                      427 non-null    int64
 2   prfInvestmentsMillNOK        427 non-null    int64
 3   prfNpdidInformationCarrier   427 non-null    int64
 4   dateSyncNPD                  427 non-null    object
dtypes: int64(3), object(2)
memory usage: 20.0+ KB
```

I hope this gives you a sense of the power behind structuring your code into modules and packages. While we've only scratched the surface, I hope it inspires you to explore more about modular programming.

5.4 Exercises

1. Expand the functionality of the `factpages` package by adding a module to work with the *Wellbore* category. This module should have functions

for extracting:

(a) The current year exploration wellbores:
descriptor: `wellbore_exploration_current_year`

(b) The last year exploration wellbores:
descriptor: `wellbore_exploration_last_year`

(c) The last 10 years exploration wellbores:
descriptor: `wellbore_exploration_last_10_years`

(d) Short list of all exploration wellbores:
descriptor: `wellbore_exploration_all_short`

(e) Long list of all exploration wellbores:
descriptor: `wellbore_exploration_all`

(f) Development wellbores:
descriptor: `wellbore_development_all`

(g) Other wellbores:
descriptor: `wellbore_other_all`

(h) CO_2 storage wellbores:
descriptor: `wellbore_co2_storage`

All these functions should allow filtering the output by fields (`wlbField` column) and completion year (`wlbCompletionYear` column).

Chapter 6

Data visualization

As discussed in Chapter 5, Python offers a user-friendly and powerful environment for data visualization. With libraries like Matplotlib, Plotly, and hvplot, creating visualizations becomes intuitive and efficient. These tools make it easy to display complex datasets, such as well logs and seismic data, as well as time series like production data. In this chapter, we'll explore these visualization techniques in greater depth, demonstrating how Python can bring subsurface data to life.

6.1 Introduction

We begin this chapter with a brief overview of plotting in Python using the Matplotlib library. This section draws on material from this excellent resource, adapted here to introduce the core concepts. To create plots, you'll need to import the matplotlib.pyplot module, which provides a simple interface for generating a wide range of visualizations. Once imported, this module allows you to plot data with just a few lines of code[1]:

```
# import libraries required for the notebook
import numpy as np # import numpy as np
import matplotlib.pyplot as plt # import matplotlib.pyplot as plt
```

```
x = np.linspace(0,5,100) # 100 evenly spaced numbers from 0 to 5
plt.plot(x, x**2) # plot x^2
plt.show() # show the plot
```

This code generates the graph in Figure 6.1a. However, if you'd like to have more control over your graph, you can plot the data like this:

[1]The code for this section is included in the notebook `chapter6_1.ipynb`

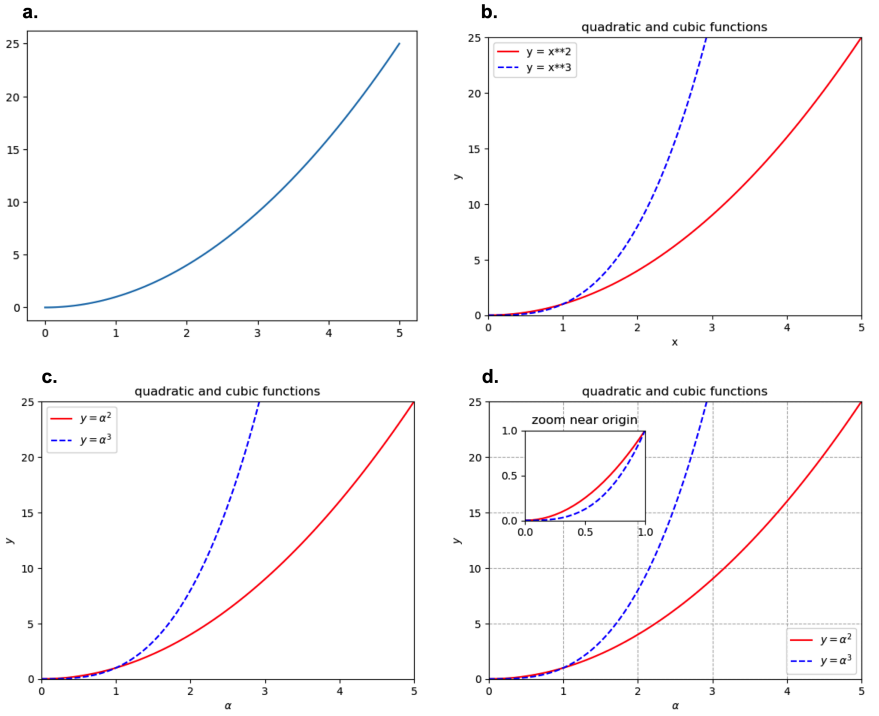


Figure 6.1: Gradual enhancement of the same plot through added elements, including labels, legends, LaTeX text, and an inset for local detail.

```
fig, ax = plt.subplots() # 1 subplot and figure and axes

ax.plot(x, x ** 2, "r-", label="y = x**2") # plot x^2 in red
ax.plot(x, x ** 3, "b--", label="y = x**3") # plot x^3 in blue
ax.legend(loc="upper left") # add legend on upper left corner
ax.set_xlabel("x") # set x label
ax.set_ylabel("y") # set y label
ax.set_xlim([0, 5]) # set x limits
ax.set_ylim([0, 25]) # set y limits
ax.set_title("quadratic and cubic functions") # set title

plt.show() # show the plot
```

This code produces the graph in Figure 6.1b. In the code above, the Matplotlib `subplots()` function creates instances of the figure (`fig`) and axes (`ax`). We can send methods to `ax` to make our graph.

What if we want to add LaTeX-formatted text to our graph? To do this, we use 'raw' text strings, which are prefixed with an `r`. Notice that the LaTeX

text is enclosed in dollar signs:

```
fig, ax = plt.subplots() # 1 subplot

ax.plot(x, x ** 2, "r-", label=r"$y = \alpha^2$") # raw text-LateX
ax.plot(x, x ** 3, "b--", label=r"$y = \alpha^3$") # raw text-LateX
ax.set_xlabel(r"$\alpha$") # raw text for LaTeX
ax.set_ylabel(r"$y$") # raw text for LaTeX
ax.legend(loc="upper left") # legend
ax.set_xlim([0, 5]) # x limits
ax.set_ylim([0, 25]) # y limits
ax.set_title("quadratic and cubic functions") # title

plt.show() # show the plot
```

This code generates the graph in Figure 6.1c.

What if we want to plot the functions in both linear and logarithmic graphs side by side? It's simple — just define the number of rows (1) and columns (2) in the `subplots()` method. You can also set the figure size using the `figsize` parameter. Since there are two subplots, there are two axes objects: `axs[0]` for the left (linear) subplot, and `axs[1]` for the right (logarithmic) subplot:

```
fig, axs = plt.subplots(1, 2, figsize=(10,4)) # 1 x 2 subplots
titles = ["normal scale", "logarithmic scale"] # titles of subplots
ymins = [0, 0.1] # min y values of subplots

for (ax, title, ymin) in zip(axs, titles, ymins):
    ax.plot(x, x ** 2, "r-", label=r"$y = \alpha^2$") # plot x^2
    ax.plot(x, x ** 3, "b--", label=r"$y = \alpha^3$") # plot x^3
    ax.grid(True, linestyle="dashed") # grid
    ax.legend(loc="upper left") # legend
    ax.set_xlabel(r"$\alpha$") # x label
    ax.set_ylabel(r"$y$") # y label
    ax.set_xlim([0, 5]) # x limits
    ax.set_ylim([ymin, 25]) # y limits
    ax.set_title(title) # title of plot

axs[1].set_yscale("log") # y axis in 2nd subplot is log
fig.tight_layout() # nice padding between subplots
plt.show() # show the plot
```

This code produces the graph in Figure 6.2a. Let's plot the functions on both normal and logarithmic scales on the same graph. To do this, we need two y-axes that share the same x-axis. The left y-axis will use a normal scale, while the right one will use a logarithmic scale. The axes `twinx()` function creates a second y-axis that shares the x-axis:

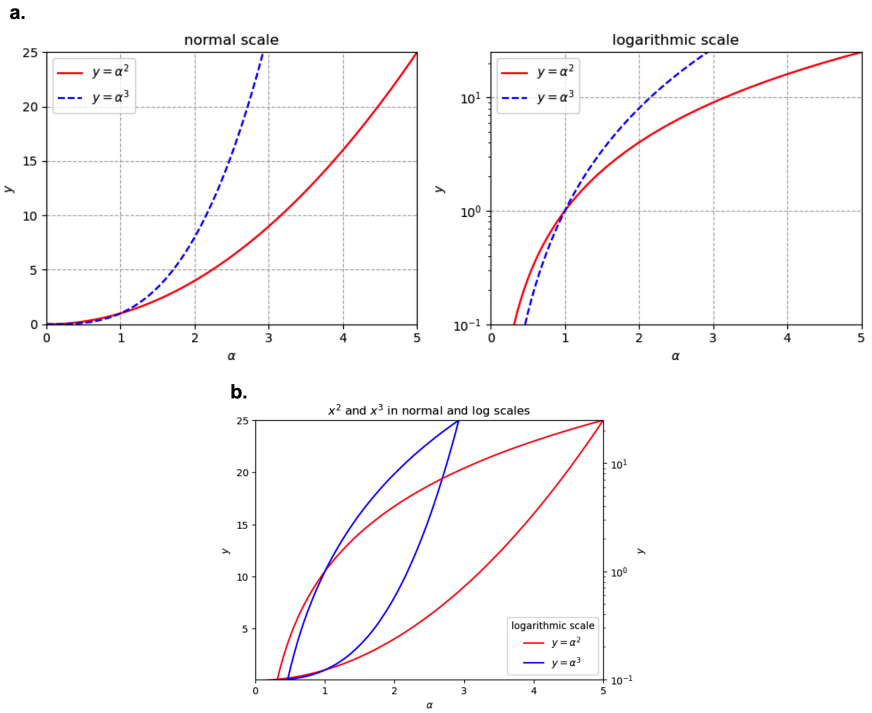


Figure 6.2: **a.** Side by side, and **b.** Overlaid linear and logarithmic plots.

```
fig, ax = plt.subplots() # 1 subplot
ax1 = ax.twinx() # add new axis sharing the x axis

ax_list = [ax, ax1] # list of axes
titles = ["normal scale", "logarithmic scale"] # legend titles
locations = ["upper left", "lower right"] # legend locations

for (ax_i, title, location) in zip(ax_list, titles, locations):
    ax_i.plot(x, x ** 2, "r-", label=r"$y = \alpha^2$")
    ax_i.plot(x, x ** 3, "b-", label=r"$y = \alpha^3$")
    ax_i.set_ylabel(r"$y$")
    ax_i.set_ylim([0.1, 25]) # y limits
    ax_i.legend(loc=location, title=title) # legend

ax1.set_yscale("log") # y axis has log scale
ax.set_xlabel(r"$\alpha$")
ax.set_xlim([0, 5]) # x limits
ax.set_title(r"$x^2$ and $x^3$ in normal and log scales")
plt.show() # show the plot
```

This code generates the graph in Figure 6.2b.

The following example demonstrates a 2×2 grid of subplots. To iterate over each subplot, we use the `ravel()` method to flatten the 2×2 `axs` array into a 1D array. Inside the loop, we use the Matplotlib `text()` function to add text to each subplot at the specified (x, y) coordinates.

```
fig, axs = plt.subplots(2, 2, figsize=(10,4))

for i, ax in enumerate(axs.ravel()):
    # plot text in the middle of the subplot
    ax.text(0.4, 0.5, f"subplot {i+1}", color = "blue")

fig.tight_layout() # nice padding between subplots
plt.show() # show the plot
```

This code produces the graph in Figure 6.3a.

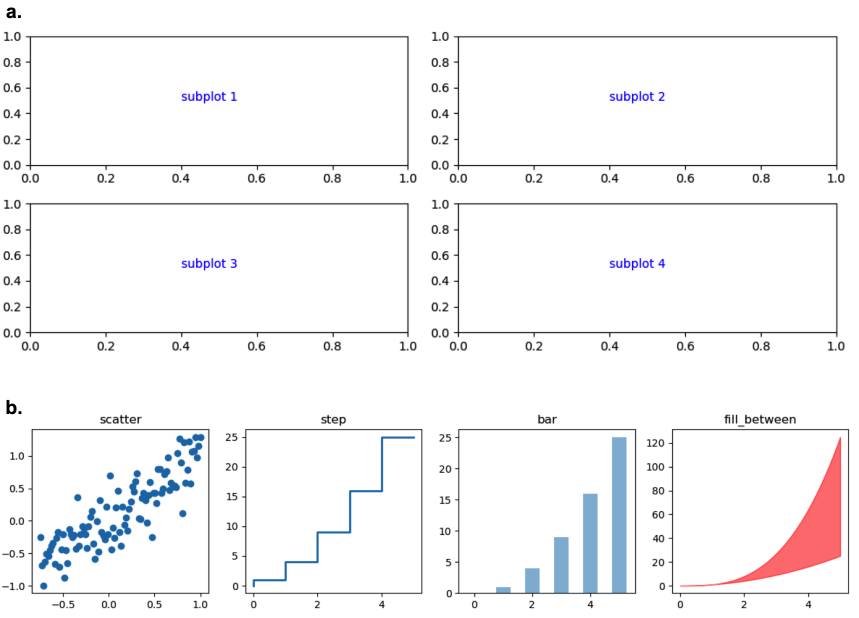


Figure 6.3: **a.** Four subplots in a 2x2 grid, and **b.** Different types of plots provided by Matplotlib.

The object oriented philosophy of the Matplotlib library is very powerful. You can control every single element of the graph. Here is an example that adds an inset to the plot of the functions so we can see how they are near the origin. Notice that in the code we use the figure `add_axes()` method to add the new inset axes:

```
fig, ax = plt.subplots() # 1 subplot

ax.plot(x, x ** 2, "r-", label=r"$y = \alpha^2$") # plot x^2
ax.plot(x, x ** 3, "b--", label=r"$y = \alpha^3$") # plot x^3
ax.grid(True, linestyle="dashed") # grid
ax.legend(loc="lower right") # legend
ax.set_xlabel(r"$\alpha$") # x label
ax.set_ylabel(r"$y$") # y label
ax.set_xlim([0, 5]) # x limits
ax.set_ylim([0, 25]) # y limits
ax.set_title("quadratic and cubic functions") # title

# To make inset, add new inset axes
inset_ax = fig.add_axes([0.2, 0.55, 0.25, 0.25]) # x, y, w and h
inset_ax.plot(x, x ** 2, "r-") # plot x^2
inset_ax.plot(x, x ** 3, "b--") # plot x^3
inset_ax.set_xlim([0, 1]) # x limits
inset_ax.set_ylim([0, 1]) # y limits
inset_ax.set_xticks([0, 0.5, 1.0]) # set x ticks
inset_ax.set_yticks([0.0, 0.5, 1.0]) # set y ticks
inset_ax.set_title("zoom near origin") # set title of inset
plt.show() # show the plot
```

This code generates the graph in Figure 6.1d. Saving the plot as an image file (e.g., png, pdf, svg, etc.) at a desired resolution (dots per inch, dpi) is easy:

```
# save graph as png and resolution = 200 dpi
fig.savefig("my_first_plot.png", dpi=200)
```

Besides the `plot` method, Matplotlib offers other functions for generating different types of plots. For a complete list of available plot types, you can check the Matplotlib plot gallery. Here are a few examples:

```
xx = np.linspace(-0.75, 1., 100) # 100 numbers from -0.75 to 1.0
n = np.array([0,1,2,3,4,5]) # another array

fig, axs = plt.subplots(1, 4, figsize=(12,3)) # 1 x 4 subplots

axs[0].scatter(xx, xx + 0.25*np.random.randn(len(xx))) # scatter
axs[0].set_title("scatter") # set title

axs[1].step(n, n**2, lw=2) # step
axs[1].set_title("step") # set title

axs[2].bar(n, n**2, align="center", width=0.5, alpha=0.5) # bar
axs[2].set_title("bar") # set title

axs[3].fill_between(x, x**2, x**3, color="red", alpha=0.5) # filled
axs[3].set_title("fill_between") # set title

fig.tight_layout() # nice padding between subplots
plt.show() # show the plot
```

This code produces the graph in Figure 6.3b.

6.2 Well logs

Let's now apply the concepts from the previous section to practical subsurface examples, starting with well logs. The file `xeek_train_subset.csv` contains log data for 12 wells in the Norwegian Continental Sector (Force 2020 ML lithology competition). Our objective is to plot the logs for well 16/10-1, as illustrated in Figure 6.4.

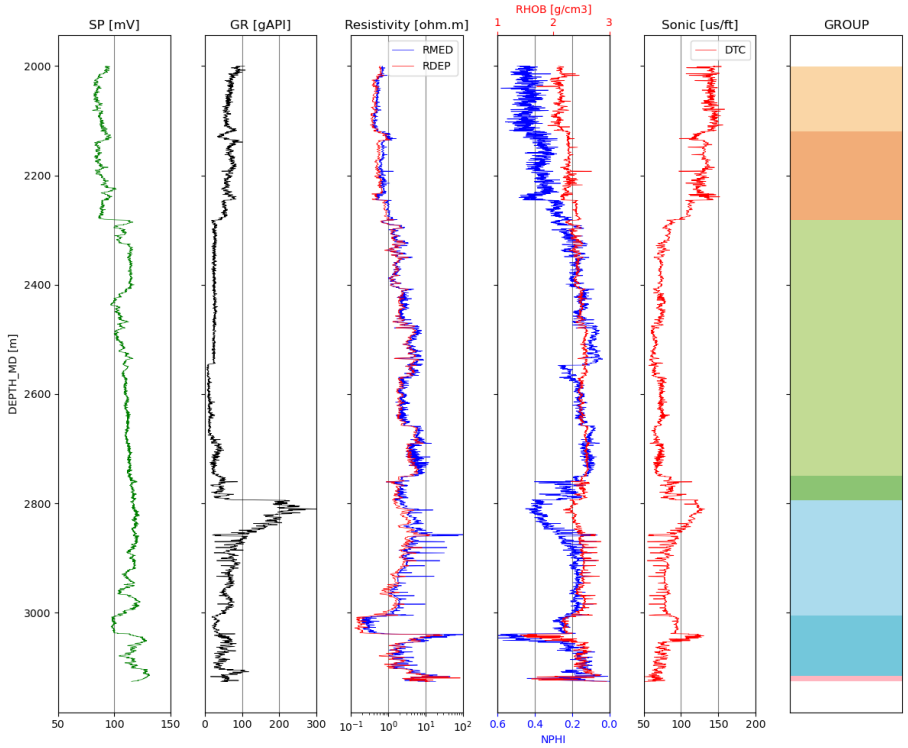


Figure 6.4: Multi-track visualization of well 16/10-1 logs using Python.

A crucial part of making any graph is preparing the data and gathering all necessary inputs. For the example shown in Figure 6.4, this process includes the following steps:

1. Extract the well data. Note that the logs are plotted from 2000 m downwards.

2. Get the group tops.

3. Get the group colors from the file `group_colors.csv`.
4. Create a figure with six side-by-side subplots.
5. Plot the logs in the first five subplots, using their respective line colors.
    - Subplots 1 and 2 show just one log curve.
    - Subplots 3 to 5 show two log curves.
    - In subplot 4, the density (`RHOB`) and neutron porosity (`NPHI`) are plotted on separate `x`-axes sharing the same `y`-axis. For the `NPHI` log, the `x`-axis should be inverted.
    - Subplot 3 requires a logarithmic `x`-axis.
6. Display the `GROUP` intervals as colored rectangles in subplot 6.

By breaking the graph creation process into sequential steps, we are essentially applying the *divide and conquer* approach. Additionally, by focusing on what needs to be done rather than how to do it, we are practicing *abstraction*.

I implemented these steps as functions in the `utilities` module of our `plot_utilities` package. The first function extracts the well data (step 1):[2]

```
def extract_well(df, well, well_col="WELL",
                 depth_col="DEPTH_MD", min_depth=None):
    """
    Extracts data for a specific well from a DataFrame.

    Parameters:
        df (pd.DataFrame): The input DataFrame containing
            data from multiple wells.
        well (str): The name of the well to extract.
        well_col (str): The name of the column that contains
            well names. Defaults to "WELL".
        depth_col (str): The name of the column that contains
            depth values. Defaults to "DEPTH_MD".
        min_depth (float, optional): If specified, only rows
            with depth >= to this value will be included.
            Defaults to None, which includes all depths.

    Returns:
        pd.DataFrame: A DataFrame containing only the rows
            corresponding to the specified well.    Returns
            an empty DataFrame if the well is not found.
    """
    # create an empty DataFrame
    well_df = pd.DataFrame()
    # if well is in the DataFrame column
    if well in df[well_col].values:
        # extract the well
```

[2]The well data is read from a CSV file. If you need to read a well in the more standard LAS format, you can use the lasio library.

```
        well_df = df[df[well_col] == well]
        # if min_depth is specified, filter the well
        if min_depth is not None:
            well_df = well_df[well_df[depth_col] >= min_depth]
    else:
        print(f"Well {well} not found in DataFrame")

    return well_df
```

The second function gets the group tops, and in fact the tops of any intervals (step 2):

```
def get_tops(df, int_col="GROUP", depth_col="DEPTH_MD"):
    """
    Identifies the top depths of intervals in a well DataFrame.

    Parameters:
        df (pd.DataFrame): The input DataFrame representing
            a single well.
        int_col (str): Name of the column containing interval
            identifiers. Defaults to "GROUP".
        depth_col (str): Name of the column containing measured
            depths. Defaults to "DEPTH_MD".

    Returns:
        dict: A dictionary mapping each interval name
            (from int_col) to its top depth (the first
            occurrence in depth_col).
    """
    # find the tops or rows where a new interval starts

    # the comparison df[int_col] != df[int_col].shift()
    # returns True where the current row's value is different
    # from the previous row. mask is a boolean Series where True
    # indicates a change in the interval column.
    mask = df[int_col] != df[int_col].shift()

    # filter the DataFrame using mask to keep only those rows
    # where the interval changes. Also, reset the index.
    m_df = df[mask].reset_index(drop=True)

    # create the dictionary of tops. Note that tops may be
    # repeated so we add the index to the key if the top is
    # repeated. Also we round the depth values to 2 decimals
    tops = {}
    for i, row in m_df.iterrows():
        top = f"{row[int_col]}"
        if top not in tops:
            tops[top] = round(row[depth_col],2)
        else:
            tops[f"{top}_{i}"] = round(row[depth_col],2)

    # add the end of the log to tops
    tops["END"] = df[depth_col].max()
```

```
    return tops
```

The third function gets the group colors, and in fact the colors of any intervals (step 3):

```
def get_colors(df1, df2, int_col="GROUP"):
    """
    Maps intervals in a well DataFrame to their corresponding
    RGB colors.

    Parameters:
        df1 (pd.DataFrame): DataFrame representing a well,
            containing interval identifiers in `int_col`.
        df2 (pd.DataFrame): DataFrame defining RGB color values
            for each interval. Must include columns:
            int_col, "RED", "GREEN", and "BLUE".
        int_col (str): Name of the column used to identify
            intervals in both DataFrames. Defaults to "GROUP".

    Returns:
        dict: A dictionary where keys are interval names and
            values are [R, G, B] color lists.
    """
    # find unique values in column
    intervals = df1[int_col].unique()

    # create the dictionary of colors
    colors = {}
    for interval in intervals:
        # find the row in df2 with the interval
        row = df2[df2[int_col] == interval]
        # get the RGB values
        if not row.empty:
            colors[interval] = [row["RED"].values[0],
                                row["GREEN"].values[0],
                                row["BLUE"].values[0]]
        # color values should be floats between 0 and 1
        # and the floats are rounded to 2 decimal places
        colors[interval] = [round(float(c/255), 2) for c in colors[interval]]
        # make sure the keys are strings
        colors[str(interval)] = colors.pop(interval)

    return colors
```

And the fourth function plots the logs (steps 5 and 6):

```
def plot_logs(df, tops, colors, axs, int_col="GROUP",
              depth_col="DEPTH_MD"):
    """
    Plots well log data with interval shading.

    Parameters:
        df (pd.DataFrame): DataFrame containing well
            log data.
```

```
        tops (dict): Dictionary mapping interval names
            to their top depths.
        colors (dict): Dictionary mapping interval names
            to RGB color lists (e.g., [R, G, B]).
        axs (list of matplotlib axes): List of subplot
            axes on which to plot the logs.
        int_col (str): Name of the column containing interval
            identifiers. Defaults to "GROUP".
        depth_col (str): Name of the column containing measured
            depths. Defaults to "DEPTH_MD".

    Description:
        The function plots the well logs in the following order:
        - SP (Spontaneous Potential)
        - GR (Gamma Ray)
        - Resistivity: RMED and RDEP
        - Density: RHOB and Neutron: NPHI
        - Sonic: DTC and DTS

        It also shades the intervals defined in `tops` using
        the corresponding colors from `colors`.

    Returns:
        None
    """
    # if not six subplots, raise an error
    if len(axs) != 6:
        print("Error: ax must have 6 subplots")
        return

    # list of logs, line colors, x limits and titles
    # for the subplots
    logs = ["SP", "GR", ["RMED","RDEP"], ["RHOB","NPHI"],
            ["DTC","DTS"], int_col]
    l_cs = ["green", "black", ["blue","red"],
                    ["red","blue"], ["red","blue"], "black"]
    x_lims = [[50, 150], [0, 300], [0.1, 100],
              [[1, 3], [0.6, 0]], [50, 200], [0, 1]]
    titles = ["SP [mV]", "GR [gAPI]", "Resistivity [ohm.m]",
              ["RHOB [g/cm3]", "NPHI"], "Sonic [us/ft]", int_col]

    # iterate over subplots and plot the logs
    for i, (log, l_c, x_lim, title) in enumerate(zip(logs, l_cs, x_lims, titles)):
        # SP and GR: i is 0 or 1, single curve each
        if i < 2:
            # plot the log
            axs[i].plot(df[log], df[depth_col], color=l_c, lw=0.5)
            if i == 0: # on first subplot
                axs[i].invert_yaxis() # y increases down
                axs[i].set_ylabel(depth_col + " [m]") # set y label
        # Resistivity and sonic: i is 2 or 4, 2 curves each
        elif i == 2 or i == 4:
            for j in range(2): # iterate over the two logs
                if not df[log[j]].isna().all(): # if data
                    # plot the log
                    axs[i].plot(df[log[j]], df[depth_col],
                                color=l_c[j], lw=0.5, label=log[j])
                    axs[i].legend(loc='upper right') # add legend
            # for resistivity x axis is log scale
            if i == 2:
                axs[i].set_xscale("log") # log scale
        # RHOB and NPHI curves: i is 3
        elif i == 3:
            ax1 = axs[i].twiny() # create a second x axis for NPHI
            ax_list = [ax1, axs[i]] # list of axes to plot on
            for j, ax in enumerate(ax_list): # iterate over axes
                # plot the log
```

```
                ax.plot(df[log[j]], df[depth_col], color=l_c[j], lw=0.5)
                # set x axis limits, labels, and colors
                ax. set_xlim(x_lim[j])
                ax.set_xlabel(title[j])
                ax.xaxis.label.set_color(l_c[j])
                ax.tick_params(axis='x', colors=l_c[j])
        # intervals: i is 6, rectangles defined by tops and colors
        elif i == 5:
            items = list(tops.items()) # list of tops items
            for j in range(len(items) - 1): # iterate over items
                key1, val1 = items[j] # interval name and top depth
                _, val2 = items[j + 1] # next interval top depth
                color_key = key1.split("_")[0] # first part of the key
                # draw a rectangle for the interval
                axs[i].axhspan(val1, val2, facecolor = colors[color_key])
            axs[i].set_xticks([]) # no x ticks for intervals

        # set x_limits, title and grid
        if i != 3: # no x_limits or title for RHOB and NPHI
            axs[i].set_xlim(x_lim)
            axs[i].set_title(title)
        if i != 5: # no grid for intervals
            axs[i].grid(axis='x', color='gray')
```

It might feel overwhelming to take in all these functions at once. If any parts are unclear, try running them step by step in separate notebook cells to better understand how they work. What's important is that we now have the tools to generate a well log graph with ease.

Let's begin by extracting the well and getting the groups tops and colors:[3]

```
# import libraries required for the notebook
import os # import os to work with directories
import pandas as pd # import pandas as pd
import matplotlib.pyplot as plt # import matplotlib.pyplot as plt
import plot_utilities as pu # import our plot utilities package
```

```
# path to well log data
path1 = os.path.join("..", "data", "xeek_train_subset.csv")
# path to group colors
path2 = os.path.join("..", "data", "group_colors.csv")

# read the well log data into pandas DataFrame
df1 = pd.read_csv(path1)
# read the group colors into pandas DataFrame
df2 = pd.read_csv(path2)

# extract well (step 1)
well = "16/10-1"
well_df = pu.extract_well(df1, well, min_depth=2000)

# get groups tops (step 2)
group_tops = pu.get_tops(well_df)
```

[3]The code for this section is included in the notebook `chapter6_2.ipynb`

```
print(group_tops)

# get groups colors (step 3)
group_colors = pu.get_colors(well_df, df2)
print(group_colors)
```

```
{'HORDALAND GP.': 2000.15, ... 'ZECHSTEIN GP.': 3116.14, 'END': 3125.8637898}
{'HORDALAND GP.': [0.98, 0.84, 0.65], ... 'ZECHSTEIN GP.': [1.0, 0.71, 0.75]}
```

Finally, we can plot the logs. The code below generates Figure 6.4.

```
# create figure and axes sharing y axis (step 4)
fig, axs = plt.subplots(1, 6, figsize=(12, 10), sharey=True)

# plot the logs (steps and 6)
pu.plot_logs(well_df, group_tops, group_colors, axs)

# present the figure
fig.tight_layout()
plt.show()
```

By structuring our code into functions, modules, and packages, we've made it highly versatile. This allows us to plot any well in the dataset or focus on specific intervals with ease. For example, to plot the facies for well 16/10-1—assuming we have a file that defines facies colors—we can do the following:

```
# rename the last second column to "FACIES"
well_df = well_df.copy()
well_df.rename(columns={well_df.columns[-2]: "FACIES"},
               inplace=True)

# path to facies colors
path3 = os.path.join("..", "data", "facies_colors.csv")

# read the facies colors into pandas DataFrame
df3 = pd.read_csv(path3)

# get facies tops (step 2)
facies_tops = pu.get_tops(well_df, int_col="FACIES")

# get facies colors (step 3)
facies_colors = pu.get_colors(well_df, df3, int_col="FACIES")

# create figure and axes sharing y axis (step 4)
fig, axs = plt.subplots(1, 6, figsize=(12, 10), sharey=True)

# plot the logs (steps 5 and 6)
pu.plot_logs(well_df, facies_tops, facies_colors, axs,
             int_col="FACIES")

# present the figure
fig.tight_layout()
plt.show()
```

This code generates Figure 6.5.

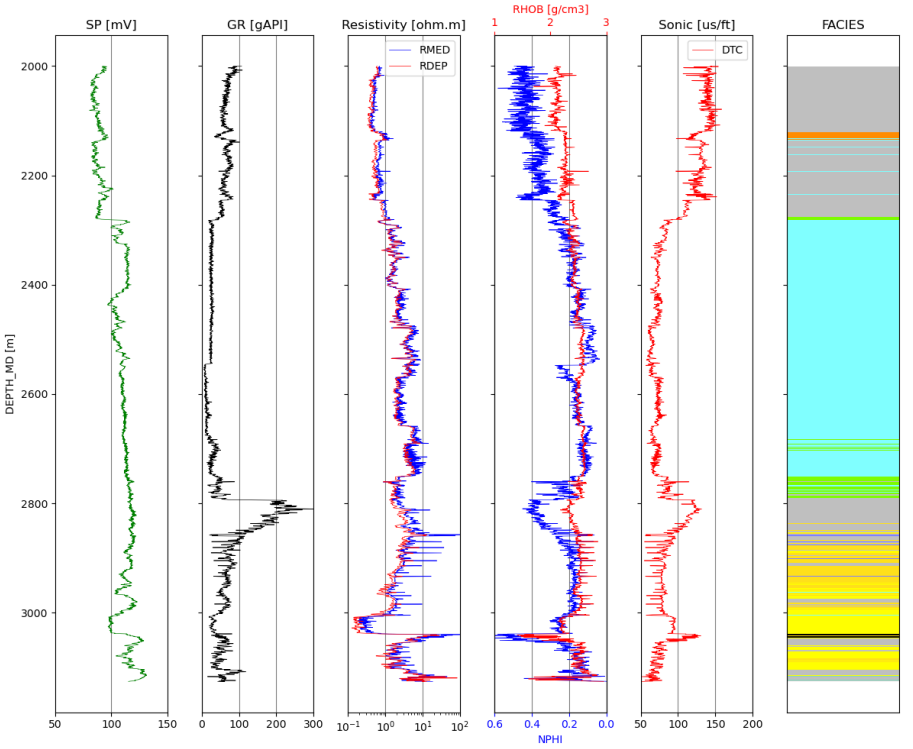


Figure 6.5: Multi-track visualization of well 16/10-1 logs using Python. The last track shows facies.

6.3 Seismic data

Plotting seismic data in Python is a straightforward process. Once loaded, the data typically takes the form of a NumPy array containing amplitude values. In this example, we'll work with a subset of the F3 seismic dataset—a 3D seismic cube acquired offshore in the Netherlands.

The `segy` file containing the seismic data is relatively large (370 MB), so it has been hosted on a remote server. To download it, we can use Python's `urllib` library. The process is straightforward: define the URL of the file and the local filename, then pass these to the `urlretrieve()` function to download and save the file in the desired directory[4].

[4]The code for this section is included in the notebook `chapter6_3.ipynb`

```
# import libraries required for the notebook
import os # import os to work with directories
from urllib import request # for downloading files
import numpy as np # import numpy as np
import matplotlib.pyplot as plt # import matplotlib.pyplot as plt
from ipywidgets import interact # import ipywidgets interact
import plotly.graph_objects as go # import plotly.graph_objects as go
from plotly.subplots import make_subplots # import make_subplots
import segyio # import segyio for reading SEG-Y files
import plot_utilities as pu # import our plot utilities package
```

```
# retrieve the data file from a remote server

# Define the remote file to retrieve
remote_url = "http://www.ux.uis.no/~nestor/Public/f3-dsmf.sgy"
# Define the local filename to save data
local_file = os.path.join("..", "data", "f3-dsmf.sgy")
# Download remote and save locally
request.urlretrieve(remote_url, local_file)
```

To read the `segy` data, we'll use the Segyio library, a Python package designed for efficient handling of SEG-Y files. We can open the file using the Segyio `open()` method. Once opened, we can access the inline, crossline, and two-way travel time (TWT) axes using the `ilines`, `xlines`, and `samples` attributes of the file object `f`, respectively. The range of each axis can be calculated by subtracting the minimum value from the maximum, and the number of elements in each can be obtained from the shape of the corresponding array.

```
# print inline, xline and twt ranges

filename = os.path.join("..", "data", "f3-dsmf.sgy") # segy file

# open segy file and print inline, xline and twt ranges
with segyio.open(filename,"r") as f:
    s = [f.ilines, f.xlines, f.samples]
    s_t = ["IL", "XL", "TWT"]
    for i in range(len(s)):
        rg = np.array([np.amin(s[i]), np.amax(s[i])])
        count = s[i].shape[0]
        print(f"{s_t[i]} range: {rg}, count: {count}")
```

```
IL range: 200 - 600, count: 401
XL range: 500 - 1100, count: 601
TWT range: 200.0 - 1500.0, count: 326
```

The seismic data spans inlines from 200 to 600, xlines from 500 to 1100, and TWT from 200 to 1500 ms. We can also determine the sampling intervals—that is, the step size between adjacent inlines, xlines, and time samples—to better understand the resolution of the dataset.

```
# inline step
il_st = (np.amax(f.ilines)
         - np.amin(f.ilines)) / (f.ilines.shape[0] - 1)
# xline step
xl_st = (np.amax(f.xlines)
         - np.amin(f.xlines)) / (f.xlines.shape[0] - 1)
# twt step
twt_st = (np.amax(f.samples)
          - np.amin(f.samples)) / (f.samples.shape[0] - 1)
# print steps
print(f"IL/XL/TWT steps: {il_st}, {xl_st}, {twt_st}")
```

```
IL/XL/TWT steps: 1.0, 1.0, 4.0
```

In this dataset, the inline and xline increments are 1, while the TWT increment is 4 ms. To read the seismic cube, we can use the Segyio `tools.cube()` method, passing the file name as input. After loading the data, we print the shape of the resulting NumPy array to confirm its dimensions along the inline, xline, and TWT axes.

```
# read seismic data
data = segyio.tools.cube(filename)

# maximum amplitude value for plotting
vmax = np.percentile(np.abs(data), 99)

# cube shape
shape = data.shape
print(f"{shape[0]} IL, {shape[1]} XL and {shape[2]} TWT samples")
```

```
401 IL, 601 XL and 326 TWT samples
```

In the code above, `vmax` is set to the 99th percentile of the absolute amplitude values. This value will be used later to scale the color range when plotting the data. Since we're working with a subset of the original 3D cube, the final step before visualization is to assign the correct inline, xline, and TWT values to the axis ticks. This is done by creating arrays of tick positions and corresponding labels for each axis.

```
il_tl = 100 # step between tick labels for inlines
xl_tl = 100 # step between tick labels for xlines
twt_tl = 200 # step between tick labels for twt

# inlines ticks positions and labels
il_pos = np.arange(0, shape[0], il_tl/il_st)
il_lab = il_pos * il_st + f.ilines[0]
il_lab = il_lab.astype(int)
print(f"il ticks = {il_pos}\n and labels = {il_lab}")
```

```
# xlines ticks positions and labels
xl_pos = np.arange(0, shape[1], xl_tl/xl_st)
xl_lab = xl_pos * xl_st + f.xlines[0]
xl_lab = xl_lab.astype(int)
print(f"xl ticks = {xl_pos}\n and labels = {xl_lab}")

# TWT ticks positions and labels
twt_pos = np.arange(0, shape[2], twt_tl/twt_st)
twt_lab = twt_pos * twt_st + f.samples[0]
twt_lab = twt_lab.astype(int)
print(f"twt ticks = {twt_pos}\n and labels = {twt_lab}")
```

```
il ticks = [  0. 100. 200. 300. 400.]
 and labels = [200 300 400 500 600]
xl ticks = [  0. 100. 200. 300. 400. 500. 600.]
 and labels = [ 500  600  700  800  900 1000 1100]
twt ticks = [  0.  50. 100. 150. 200. 250. 300.]
 and labels = [ 200  400  600  800 1000 1200 1400]
```

We can now plot the data. To facilitate this process, I have made two functions in the `utilities` module of our `plot_utilities` package. The first function plots a trace:

```
def plot_trace(trace, y, vmax, twt_pos, twt_lab, ax):
    """
    Plot trace of the seismic data.

    Input:
        trace: 1D numpy array with the trace.
        y: y axis values.
        vmax: max value for the x axis.
        twt_pos: positions of the time values on the y axis.
        twt_lab: labels for the time values on the y axis.
        ax: axis of the subplot.
    """
    # plot the trace
    ax.plot(trace, y, color="black")

    # add grid
    ax.grid()

    # set the x limits
    ax.set_xlim(-vmax, vmax)

    # invert the y axis so that time increases downwards
    ax.invert_yaxis()
    # set the y limits
    ax.set_ylim(y[-1], y[0])
    # set the y ticks and labels
    ax.set_yticks(twt_pos)
    ax.set_yticklabels(twt_lab)

    # set axes labels
```

```
    ax.set_xlabel("Amplitude")
    ax.set_ylabel("Time [ms]")
```

And the second function plots a slice, which can be either an inline, xline, or time slice:

```
def plot_slice(slice, vmax, title, sl_type,
               ax_pos, ax_lab, fig, ax, cb=True):
    """
    Plot a slice of the seismic data.

    Input:
        slice: 2D array with the seismic data.
        vmax: max value for the color scale.
        title: title of the plot.
        sl_type: slice type (inline, xline or time).
        ax_pos: positions of the axes.
        ax_lab: labels for the axes.
        fig: figure object.
        ax: axis of the subplot.
        cb: boolean for colorbar.
    """
    slice_plot = slice.copy()
    # if inline or xline slice, transpose the slice
    if sl_type == "inline" or sl_type == "xline":
        slice_plot = slice_plot.T

    # plot the slice
    sim = ax.imshow(slice_plot, cmap="RdBu",
                    vmin=-vmax, vmax=vmax, aspect="auto")
    # colorbar
    if cb:
        fig.colorbar(sim, ax=ax)

    # invert axis if k slice
    if sl_type == "time":
        ax.invert_yaxis()

    # set title
    ax.set_title(title)

    # set x label
    if sl_type == "xline":
        ax.set_xlabel("Inline")
    elif sl_type == "inline" or sl_type == "time":
        ax.set_xlabel("Xline")
    # set x ticks
    ax.set_xticks(ax_pos[0])
    ax.set_xticklabels(ax_lab[0])

    # set y label
    if sl_type == "time":
        ax.set_ylabel("Inline")
    elif sl_type == "inline" or sl_type == "xline":
        ax.set_ylabel("Time [ms]")
```

```
    # set y ticks
    ax.set_yticks(ax_pos[1])
    ax.set_yticklabels(ax_lab[1])
```

In the `plot_slice()` function above, we use the Matplotlib `imshow()` method to visualize the seismic data as a pseudocolor image. Let's start by plotting one trace of the seismic data at inline 400 and xline 800:

```
# plot trace

il, xl = 400, 800 # inline and xline

il_id = int((il - f.ilines[0]) / il_st) # inline index

xl_id = int((xl - f.xlines[0]) / xl_st) # xline index

trace = data[il_id, xl_id, :] # get trace data

time_id = np.arange(shape[2]) # time index

# create figure
fig, ax = plt.subplots(figsize=(3, 7)) # 1 subplot

# plot trace
pu.plot_trace(trace, time_id, vmax, twt_pos, twt_lab, ax)
```

This code produces the plot in Figure 6.6a.

Now, let´s plot a slice, for example inline 350:

```
# plot slice

# slice type: inline, xline or time
sl_type = "inline"

# slice value: allowed values depend on the slice type
# for inline the value should be between 200 and 600,
# for xline between 500 and 1100,
# and for time between 200 and 1500 ms
value = 350

# slice and axes ticks positions and labels
if sl_type == "inline":
    index = int((value - f.ilines[0]) / il_st) # inline index
    slice = data[index, :, :]
    ax_pos = [xl_pos, twt_pos]
    ax_lab = [xl_lab, twt_lab]
    title = f"{sl_type} {value}"
elif sl_type == "xline":
    index = int((value - f.xlines[0]) / xl_st)
    slice = data[:, index, :]
    ax_pos = [il_pos, twt_pos]
```

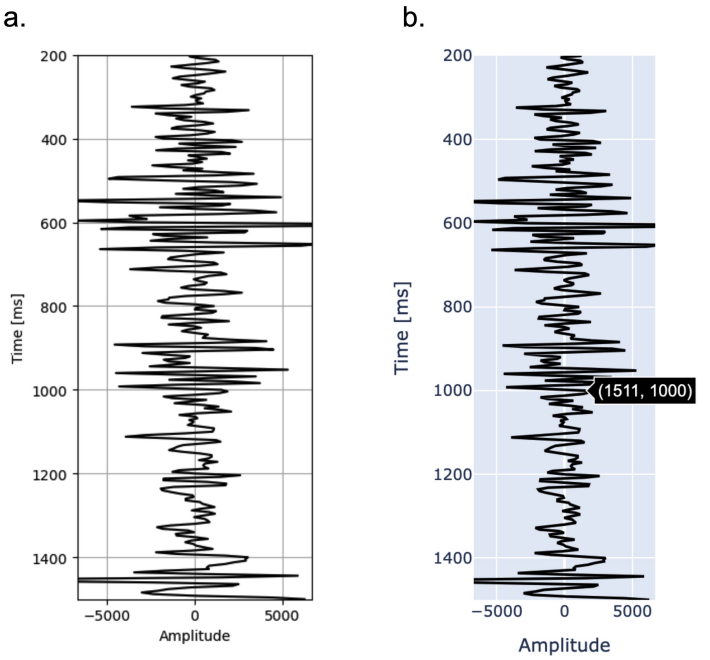


Figure 6.6: F3 seismic trace at inline 400 and xline 800, visualized using **a.** Matplotlib and **b.** Plotly.

```
    ax_lab = [il_lab, twt_lab]
    title = f"{sl_type} {value}"
elif sl_type == "time":
    index = int((value - f.samples[0]) / twt_st)
    slice = data[:, :, index]
    ax_pos = [xl_pos, il_pos]
    ax_lab = [xl_lab, il_lab]
    title = f"{sl_type} {value} ms"

# create figure
fig, ax = plt.subplots(figsize=(8, 6)) # 1 subplot

# plot slice
pu.plot_slice(slice, vmax, title, sl_type,
              ax_pos, ax_lab, fig, ax, cb=False)

plt.show() # show plot
```

This code generates the plot in Figure 6.7. The code is quite flexible. You can change the slice type (`sl_type`) and/or the slice value (`value`) to visualize another slice. For example, try visualizing xline 700, and time slice 1000.

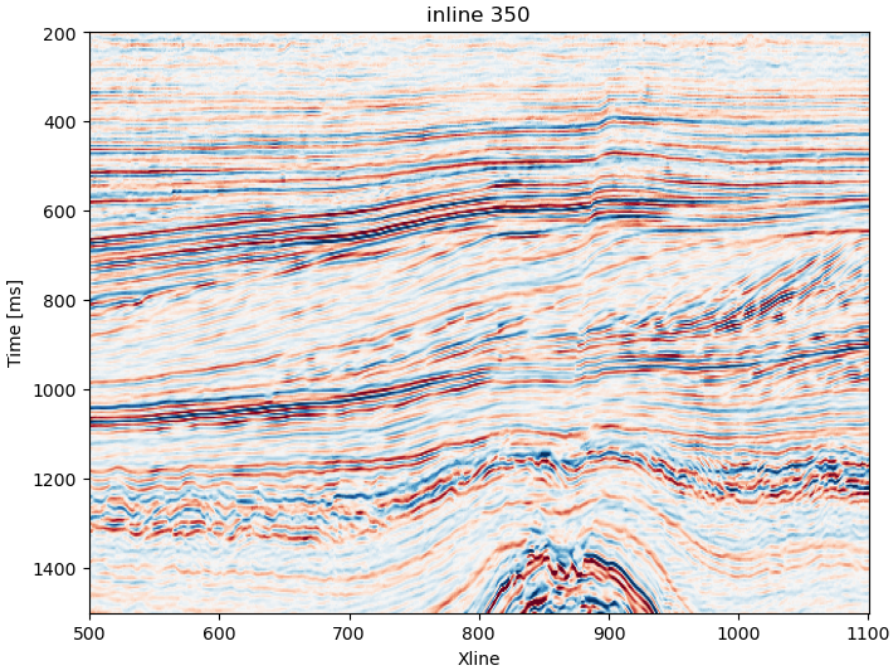


Figure 6.7: Inline 350 from the F3 dataset, visualized using Matplotlib.

Next, we'll use ipywidgets to interactively explore the data, allowing us to change the inline, xline and time slices dynamically. ipywidgets, also referred to as Jupyter widgets or simply widgets, are interactive HTML elements designed for use in Jupyter notebooks. By defining a plotting function and associating it with a widget, we can dynamically adjust the slices using slider buttons:

```
def update_plot(il, xl, time):
    """
    updates the plot with the selcted inline, xline
    and time values
    """
    # indexes for inline, xline and time
    il_id = int ((il - f.ilines[0]) / il_st)
    xl_id = int ((xl - f.xlines[0]) / xl_st)
    time_id = int ((time - f.samples[0]) / twt_st)

    # create figure
    fig, ax = plt.subplots(2,2,figsize=(14,12))

    # lists to facilitate plotting
    sl_types = ["inline", "xline", "time"]
    slices = [data[il_id, :, :], data[:, xl_id, :],
              data[:, :, time_id]]
```

```
    ax_pos = [[xl_pos, twt_pos], [il_pos, twt_pos],
              [xl_pos, il_pos]]
    ax_lab = [[xl_lab, twt_lab], [il_lab, twt_lab],
              [xl_lab, il_lab]]
    titles = [f"Inline {il:.0f}", f"Xline {xl:.0f}",
              f"Time {time:.0f} ms"]
    lines = [[xl_id, time_id], [il_id, time_id],
             [xl_id, il_id]]

    # plot slices
    for i, sl_type in enumerate(sl_types):
        # plot slice
        pu.plot_slice(slices[i], vmax, titles[i], sl_type,
                      ax_pos[i], ax_lab[i], fig,
                      ax[i//2, i%2], cb=False)
        # plot lines
        ax[i//2, i%2].axvline(lines[i][0], color='k',
                              linestyle='--', linewidth=2)
        ax[i//2, i%2].axhline(lines[i][1], color='k',
                              linestyle='--', linewidth=2)

    # four subplot is not used
    ax[1, 1].axis('off')

    # present figure
    fig.tight_layout()
    plt.show()

# interact with the function
interact(update_plot,
         il=(np.amin(f.ilines), np.amax(f.ilines), il_st),
         xl=(np.amin(f.xlines), np.amax(f.xlines), xl_st),
         time=(np.amin(f.samples), np.amax(f.samples), twt_st));
```

This code produces the plot shown in Figure 6.8. The first slider allows us to select the inline, the second slider the xline, and the third slider the time slice. The black dashed lines show the location of the slices.

6.4 Interactive plots

ipywidgets allow us to dynamically adjust the input parameters of a plot, but the plot itself remains static — moving the cursor over it doesn't reveal any information, and we cannot zoom or pan. To add this kind of interactivity, we can use the powerful Plotly library. Plotly offers extensive capabilities for creating interactive 2D and 3D plots, as well as maps. In this section, we will provide a brief introduction to its features.

Let's begin by plotting a trace from our seismic cube. For this, we use the Plotly `graph_objects` module (imported as `go`).[5]

[5]Plotly also offers the `express` module, which is easier to use but provides fewer options for customizing plots.

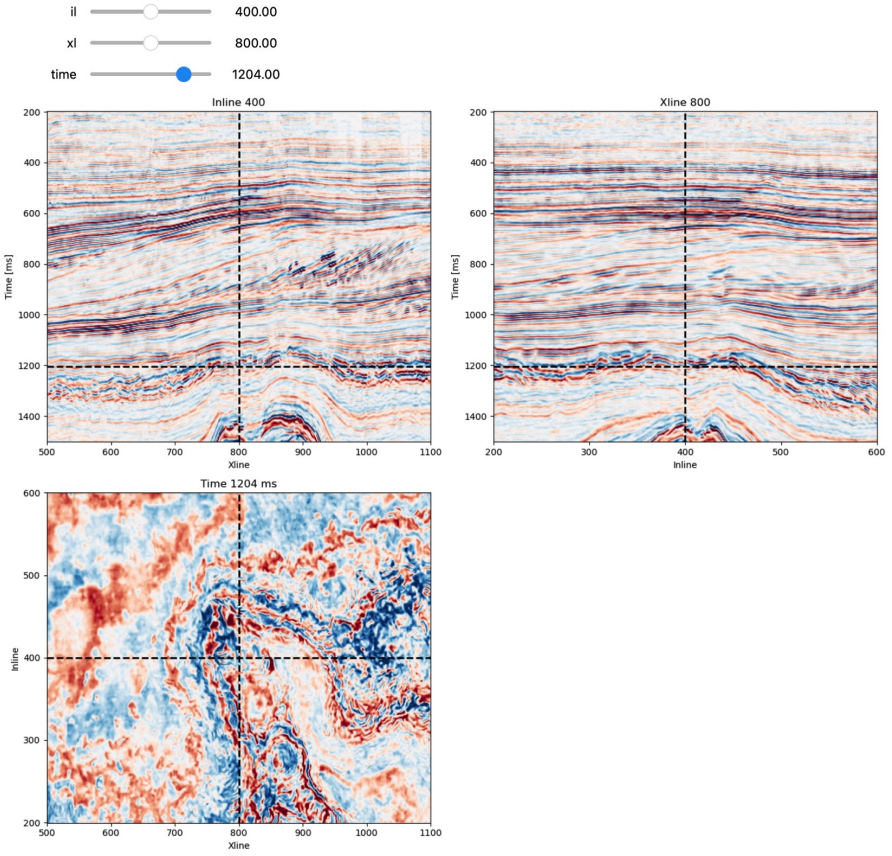


Figure 6.8: Controlling dynamically the displayed inline, xline and time slice of a seismic cube via widgets.

```
fig = go.Figure() # create figure

# convert time index to time values
time = time_id * twt_st + f.samples[0]

# add trace to the figure
fig.add_trace(go.Scatter(x=trace, y=time,
                         mode="lines",
                         line=dict(color="black")))

# set x and y axis
fig.update_xaxes(range=[-vmax, vmax],
                 title_text="Amplitude")
fig.update_yaxes(range=[time[-1], time[0]],
                 title_text="Time [ms]")
```

```
# set figure size
fig.update_layout(width=300, height=600)

fig.show() # show the figure
```

This code generates Figure 6.6b. Notice that as you hover the cursor over the plot, the amplitude and time values are displayed.

Now, let's plot the trace alongside the time slice. To create two subplots — one for the trace and another for the time slice — we use the `make_subplots()` method from the Plotly `subplots` module:

```
# Extract a time slice
value = 1200
index = int((value - f.samples[0]) / twt_st) # time index
slice = data[:, :, index]

# Create subplots
fig = make_subplots(rows=1, cols=2,
                    subplot_titles=(None, f"Time = {value} ms"))

# elements to draw
el_defs = [
    # trace on 1st subplot
    (go.Scatter(x=trace, y=time, mode="lines", name="Trace",
                line_color="black"), 1, 1),
    # slice on 2nd subplot
    (go.Heatmap(z=slice, colorscale="RdBu", name="Amplitude",
                showscale=False, zmin=-vmax, zmax=vmax), 1, 2),
    # trace location on 2nd subplot
    (go.Scatter(x=[xl_id], y=[il_id], mode="markers", name="Trace",
                marker=dict(size=10, color="black")), 1, 2)
]
# add to the figure
for el, row, col in el_defs:
    fig.add_trace(el, row=row, col=col)

# update axes
fig.update_xaxes(title_text="Amplitude",
                 row=1, col=1, range=[-vmax, vmax])
fig.update_yaxes(title_text="Time [ms]",
                 row=1, col=1, autorange="reversed")
xl_text = [str(v) for v in xl_lab]
fig.update_xaxes(title_text="Xline", tickvals=xl_pos,
                 ticktext=xl_text, row=1, col=2)
il_text = [str(v) for v in il_lab]
fig.update_yaxes(title_text="Inline", tickvals = il_pos,
                 ticktext=il_text, row=1, col=2)

# figure size and legend off
fig.update_layout(width=900, height=600, showlegend=False)

fig.show()
```

This code produces the interactive plot in Figure 6.9.

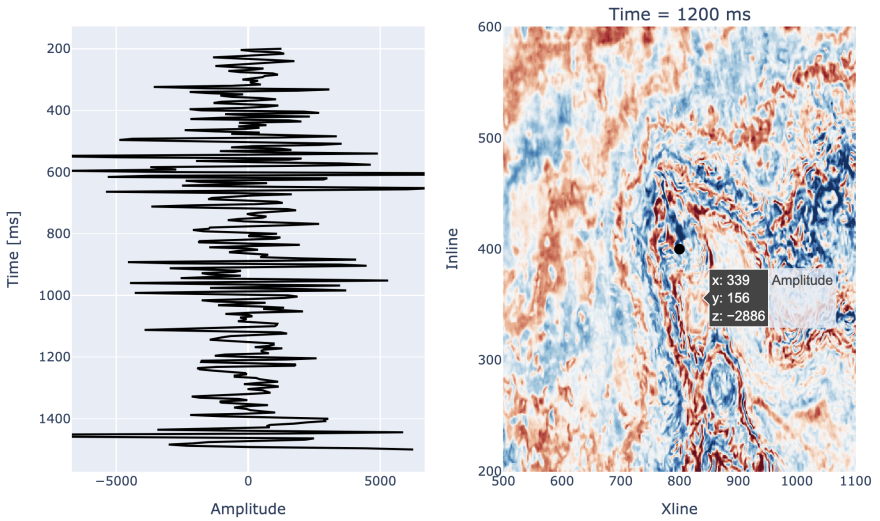


Figure 6.9: F3 seismic data visualized using Plotly, with **a.** the trace and **b.** the time-slice shown side-by-side. Notice that amplitude values can be extracted by hovering the cursor over the slice.

We have only begun to explore what Plotly offers. With a bit of practice, you'll find it a powerful tool for creating clear and interactive visualizations. I encourage you to experiment further and discover how it can support your own projects.

6.5 Plotting DataFrames

DataFrames in Pandas come with built-in plotting capabilities that make it easy to visualize data directly. Under the hood, a DataFrame acts as a lightweight wrapper around Matplotlib's `pyplot` interface, allowing you to quickly create a variety of plots — like line graphs, bar charts, and area plots — using simple method calls such as `plot()`. This makes it convenient to explore and present data without needing to manually set up a full Matplotlib workflow.

Let's look at an example using the Factpages data on monthly field production from the NCS. We'll start by reading the data with our `factpages` package[6]:

[6]The code for this section is included in the notebook `chapter6_4.ipynb`

```
# import libraries required for the notebook
import pandas as pd # import pandas as pd
import hvplot.pandas # import hvplot to plot DataFrames
import factpages as fp # import our Factpages package
```

```
# read monthly production data of ConocoPhillips fields
fields = ["EKOFISK", "ELDFISK", "TOMMELITEN GAMMA",
          "TOR", "VEST EKOFISK", "ALBUSKJELL", "VALHALL",
          "HOD", "TOMMELITEN A"]

field = fp.Field() # create field object

df = field.monthly_production(fields) # read data

df.head() # show first 5 rows
```

	prfInformationCarrier	prfYear	prfMonth	prfPrdOilNetMillSm3	prfPrdGasNetBillSm3	prfPrdNGLN
102	ALBUSKJELL	1979	5	0.00417	0.00768	
103	ALBUSKJELL	1979	6	0.01164	0.02090	
104	ALBUSKJELL	1979	7	0.03491	0.05398	
105	ALBUSKJELL	1979	8	0.09623	0.12506	
106	ALBUSKJELL	1979	9	0.11020	0.13332	

Before we can plot the data, we need to do some preparation. The code below:

1. Renames the columns to more meaningful names, including `year` and `month`.
2. Creates a `Date` column by combining the `year` and `month` columns.
3. Drops unnecessary columns.
4. Sets the `Date` column as the index— this is important for plotting the production data over time.
5. Checks the result.

```
# Make a full copy first to avoid SettingWithCopyWarning
df = df.copy()

# get the columns names
columns = df.columns.to_list()

# Rename columns
df.rename(columns={columns[1]:"year", columns[2]:"month",
                   columns[3]:"Oil MSm3", columns[4]:"Gas BSm3",
                   columns[5]:"NGL MSm3", columns[6]:"Condensate MSm3",
                   columns[7]:"Oil eq. MSm3", columns[8]:"Water MSm3"},
```

```
                    inplace=True)

# Make a Date column from the year and month columns
df["Date"] = pd.to_datetime(df[["year", "month"]].assign(day=1))

# Drop the unnecessary columns
df.drop(columns=["year", "month", columns[9]], inplace=True)

# Set the Date column as index
df.set_index("Date", inplace=True)

# Check the result
df.head()
```

Date	prfInformationCarrier	Oil MSm3	Gas BSm3	NGL MSm3	Condensate MSm3	Oil eq. MSm3	Water MSm3
1979-05-01	ALBUSKJELL	0.00417	0.00768	0.00033	0.0	0.01218	0.0
1979-06-01	ALBUSKJELL	0.01164	0.02090	0.00094	0.0	0.03347	0.0
1979-07-01	ALBUSKJELL	0.03491	0.05398	0.00447	0.0	0.09336	0.0
1979-08-01	ALBUSKJELL	0.09623	0.12506	0.01204	0.0	0.23333	0.0
1979-09-01	ALBUSKJELL	0.11020	0.13332	0.01188	0.0	0.25539	0.0

We now have the data in the form we need. Let's plot the production from the Ekofisk field — impressively, just a few lines of code will do the job and generate Figure 6.10:

```
# extract data for a specific field
field = "EKOFISK"
df_field = df[df["prfInformationCarrier"] == field]

# plot the production data
axs = df_field.plot(kind="area",subplots=True, figsize=(12,9));

# y axes limits
max_value = df_field["Oil eq. MSm3"].max() * 1
for ax in axs:
    ax.set_ylim(0, max_value)
```

While the basic `plot()` method gives us quick and useful plots, we can create even better and interactive visualizations using hvplot, a powerful library that extends DataFrame plotting with rich, interactive features. Let's plot the Ekofisk production data using hvplot:

```
# line plot with hvplot
```

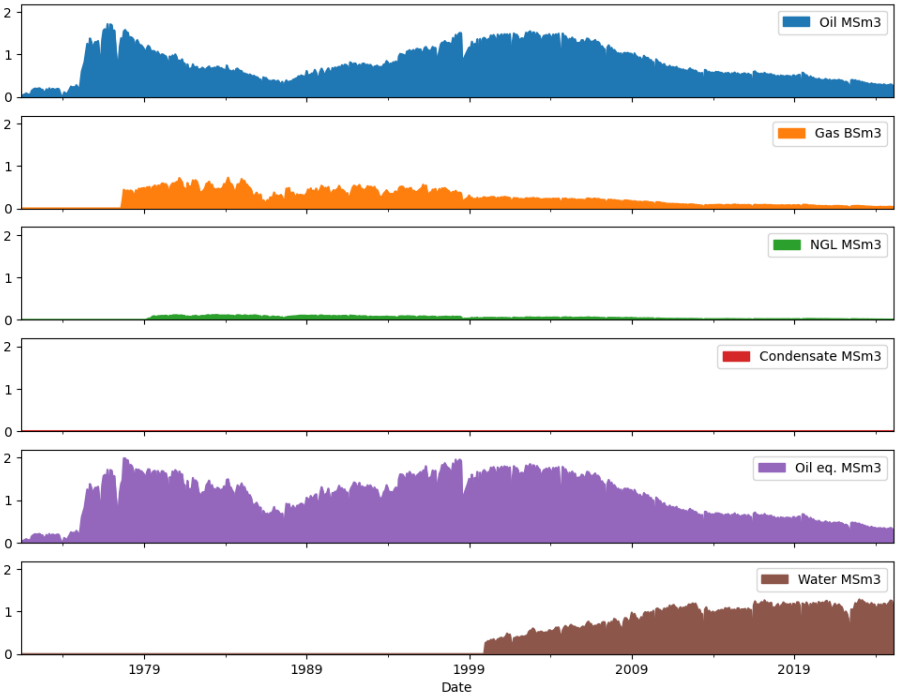


Figure 6.10: Ekofisk monthly production as visualized using the DataFrame plot() method.

```
# names of columns to plot
columns = df_field.columns.to_list()
columns.pop(0) # remove the first element

df_field.hvplot.line(x="Date", y=columns,
                     title=f"Production data for {field}",
                     width=800, height=400).opts(ylim=(0, max_value))
```

This code produces Figure 6.11a. Notice how the data values are displayed when you hover the cursor over the curves. You can also toggle the curves on and off by clicking the legend entries.

Creating an area plot is just as straightforward:

```
# area plot
is_stacked = False # stacked area or not

if is_stacked:
    y_max = max_value * 2
    alpha = 1.0
```

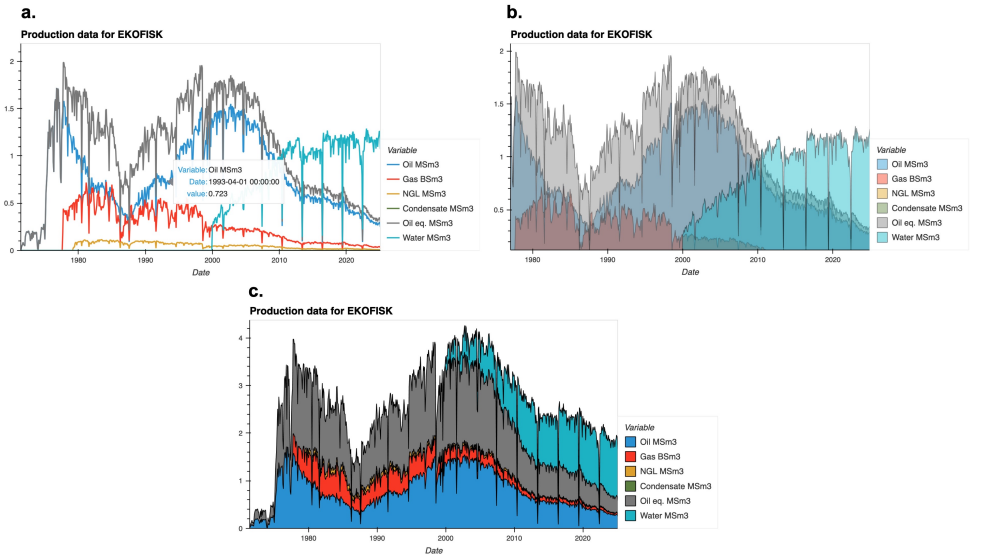


Figure 6.11: Ekofisk monthly production data visualized using the **a.** hvplot line() method, and the area() method with **b.** non-stacked and **c.** stacked data.

```
else:
    y_max = max_value
    alpha = 0.4

df_field.hvplot.area(x="Date", y=columns,
                     stacked=is_stacked, alpha=alpha,
                     title=f"Production data for {field}",
                     width=800, height=400).opts(ylim=(0, y_max))
```

This code generates a non-stacked area plot (Figure 6.11b). Setting `is_stacked` to `True` produces a stacked area plot instead (Figure 6.11c).

The hvplot library is incredibly powerful, offering an excellent way to create high-quality, interactive graphs directly from DataFrames. It's definitely worth exploring for anyone looking to enhance their data visualizations.

6.6 Exercises

1. Modify the `plot_logs()` function in the `utilities` module to:

    - Allow the function to accept a list of two log names (e.g., `["PEF", "CALI"]`) to be plotted in the first two subplots. This makes possible to visualize any logs of interest in these subplots.

- Accept a list of x-axis limits for each log.

2. Modify the `update_plot()` function in notebook `chapter6_3.ipynb` to include four subplots. The fourth subplot should display the trace at the selected inline and xline.

3. The file `xeek_train_subset.csv` contains the east coordinates (`X_LOC`) and north coordinates (`Y_LOC`) of the 12 wells included in the file. Plot the wells on a map, showing both their locations and names.

4. Seismic sections often exaggerate the vertical scale relative to the horizontal scale. This exaggeration is quantified by the vertical exaggeration factor, V. This factor affects bedding dip (δ) and thickness (t) as follows:

$$\tan \delta' = V \tan \delta$$

$$\frac{t'}{t} = \frac{\sin \delta'}{\sin \delta}$$

where δ' and t' are the exaggerated bedding dip and thickness, respectively.

   (a) Plot unexaggerated dip (δ, in degrees) versus exaggerated dip (δ', in degrees) for $V = 0.5$, 1, 2, 3, 4, 6 and 10.

   (b) Plot unexaggerated dip (δ, in degrees) versus normalised exaggerated thickness (t'/t) for $V = 0.5$, 1, 2, 3, 4, 6 and 10.

Chapter 7

Data analysis

Data analysis is the process of exploring, understanding, and drawing insights from data. In this chapter, we will walk through essential steps in analyzing data, starting from examining individual variables (univariate analysis) to exploring relationships between multiple variables (multivariate analysis). We'll also learn how to group similar data points using a technique called K-Means classification.

An important part of data analysis is *curating the data*—cleaning, selecting, and preparing it for meaningful insights. We began this process in the previous chapter, and we will continue doing it in the practical examples throughout this chapter.

7.1 Univariate analysis

Univariate analysis is the simplest form of data analysis, where we examine one variable at a time. The goal is to understand the basic properties of that variable—such as its distribution, central tendency (like mean or median), and spread (like range or standard deviation). This type of analysis helps us get a clear picture of each variable on its own before looking at how variables relate to each other.

To illustrate this, we analyze a subset of wells from the Force 2020 ML competition, provided in the file `xeek_train_subset.csv`. We begin by loading the dataset along with `lith_colors.csv`, which defines the color scheme for the lithologies. Using a dictionary, we translate lithology codes into lithology names and add a corresponding `LITH` column to the DataFrame. Finally, we use our `plot_utilities` package to generate a dictionary mapping each lithology to its associated color.[1]

[1]The code for this chapter is included in the notebook `chapter7.ipynb`

```
# Import libraries required for the notebook
import os # import os to work with directories
import pandas as pd # import pandas as pd
import matplotlib.pyplot as plt # import matplotlib.pyplot as plt
import seaborn as sns # import seaborn as sns
import plot_utilities as pu # import our plot utilities package
import data_analysis as da # import our data analysis package
```

```
# paths to files
path_1 = os.path.join("..", "data", "xeek_train_subset.csv")
path_2 = os.path.join("..", "data", "lith_colors.csv")

# read the files into pandas dataframes
df_1 = pd.read_csv(path_1) # well logs
df_2 = pd.read_csv(path_2) # facies color codes

# make a dictionary of lithology numbers and lithology names
lithology_dict = {30000: 'Sandstone', 65030: 'SS/Shale',
                  65000: 'Shale', 80000: 'Marl',
                  74000: 'Dolomite', 70000: 'Limestone',
                  70032: 'Chalk', 88000: 'Halite',
                  86000: 'Anhydrite', 99000: 'Tuff',
                  90000: 'Coal', 93000: 'Basement'}

# add lithology names to the dataframe, based on the second
# last column (which contains lithology numbers)
df_1["LITH"] = df_1[df_1.columns[-2]].map(lithology_dict)

# get the lithology colors
colors = pu.get_colors(df_1, df_2, int_col="LITH")
print(colors)
```

```
{'Shale': [0.75, 0.75, 0.75], 'Sandstone': [1.0, 1.0, 0.0], 'SS/Shale': [1.0, 0.
    88, 0.1], 'Limestone': [0.5, 1.0, 1.0], 'Tuff': [1.0, 0.55, 0.0], 'Marl': [0
    .49, 0.99, 0.0], 'Anhydrite': [1.0, 0.5, 1.0], 'Dolomite': [0.5, 0.5, 1.0],
    'Chalk': [0.5, 1.0, 1.0], 'Coal': [0.0, 0.0, 0.0], 'Halite': [0.49, 0.87, 0.
    75]}
```

Let's analyze one variable in the dataset, for example gamma ray (`GR`). To begin, we can create a table where each column represents a lithology, and each row shows a statistical summary of `GR` values within that lithology—including the count, minimum, maximum, mean, standard deviation, and key percentiles. The following function, available in the `analysis` module of our `data_analysis` package, generates this summary:

```
def property_table(df, class_col, prop_col):
    """
    For a given DataFrame df with well logs, this function
    creates a table with the statistics of a property prop_col
    for each class in class_col.
```

```
    Input:
    df: DataFrame with the well logs
    class_col: string with the name of the column with the
        classes (e.g., "LITH" for lithology)
    prop_col: string with the name of the column with the
        property to analyze (e.g., "GR" for gamma ray)

    Output:
    table: DataFrame with the statistics of the property
        for each class
    """
    # get the classes
    classes = df[class_col].unique()
    # create a table
    table = pd.DataFrame(index=["count", "min",
                                "max", "mean", "std",
                                "25%", "50%", "75%"])
    for clas in classes:
        # get the porosity for the flow unit
        prop = df[df[class_col] == clas][prop_col]
        # add statistics to the table
        table[clas] = [prop.count(), prop.min(), prop.max(),
                       prop.mean(), prop.std(),
                       prop.quantile(0.25), prop.median(),
                       prop.quantile(0.75)]

    return table
```

We can use this function to generate the statistical summary of GR in the different lithologies:

```
# GR statistics for the different lithologies
table = da.property_table(df_1, "LITH", "GR")
table.head(10)
```

	Shale	Sandstone	SS/Shale	Limestone	Tuff	Marl	Anhydrite	Dolo
count	82390.000000	15794.000000	6573.000000	11912.000000	1498.000000	4968.000000	374.000000	188.00(
min	7.870507	10.610555	23.635189	5.951932	20.513954	8.058441	1.771702	5.15
max	804.298950	645.004517	184.773926	169.651474	102.156212	131.953079	98.322258	96.89
mean	87.182635	48.239409	73.928355	28.557860	51.412686	56.720305	13.719988	40.33
std	32.570874	34.044555	28.057375	18.444981	17.999224	23.362652	15.857027	18.37
25%	63.819340	30.609406	53.699749	14.334978	38.081333	39.522079	3.543179	22.43
50%	81.913429	38.792194	64.733566	22.451968	49.374956	53.066145	4.824040	39.10
75%	105.347116	50.467609	85.839676	39.955466	61.885165	75.008862	24.987844	53.97

While this summary table is useful, a graphical display can make it easier to understand the distribution of `GR` values. The following function from our `analysis` module plots the property (`GR`) for each class (`LITH`) using either box plots or violin plots. Note that this and several functions in this chapter rely on the Seaborn library (imported as `sns`), which provides a high-level interface for creating statistical graphics in Python:

```
def property_plot(df, class_col, colors, prop_col,
                  type="boxplot"):
    """
    For a given DataFrame df with well logs, this function
    plots the property prop_col for each class in class_col.

    Input:
    df: DataFrame with the well logs
    class_col: string with the name of the column with the
        classes (e.g., "LITH" for lithology)
    colors: dictionary with the class names as the keys
        and the colors as [red, green, blue] values
    prop_col: string with the name of the column with the
        property to analyze (e.g., "GR" for gamma ray)
    type: string with the type of plot to create.
        Options are "boxplot" and "violin"

    Output:
        returns the figure
    """
    # create a figure
    fig, ax = plt.subplots(figsize=(10, 6))
    # create the plot
    if type == "boxplot":
        # create a boxplot
        sns.boxplot(x=class_col, y=prop_col, data=df,
                    hue=class_col, palette=colors, ax=ax)
    elif type == "violin":
        # create a violin plot
        sns.violinplot(x=class_col, y=prop_col, data=df,
                       hue=class_col, palette=colors, ax=ax)
    else:
        raise ValueError("type must be 'box' or 'violin'")

    # set the x-axis label
    ax.set_xlabel(class_col)
    # set the y-axis label
    ax.set_ylabel(prop_col)
    # show the plot
    plt.show()

    return fig
```

Let's use this function to visualize the distribution of `GR` in the different lithologies using a box plot:

```
# GR in the different lithologies as box plot
fig = da.property_plot(df_1, "LITH", colors, "GR")
```

This code generates Figure 7.1a. The box plot is a graphical tool that shows how values are spread out in a dataset. It highlights the median (the middle value), the range of most values (the interquartile range), and any values that are unusually high or low—these are called *outliers*.

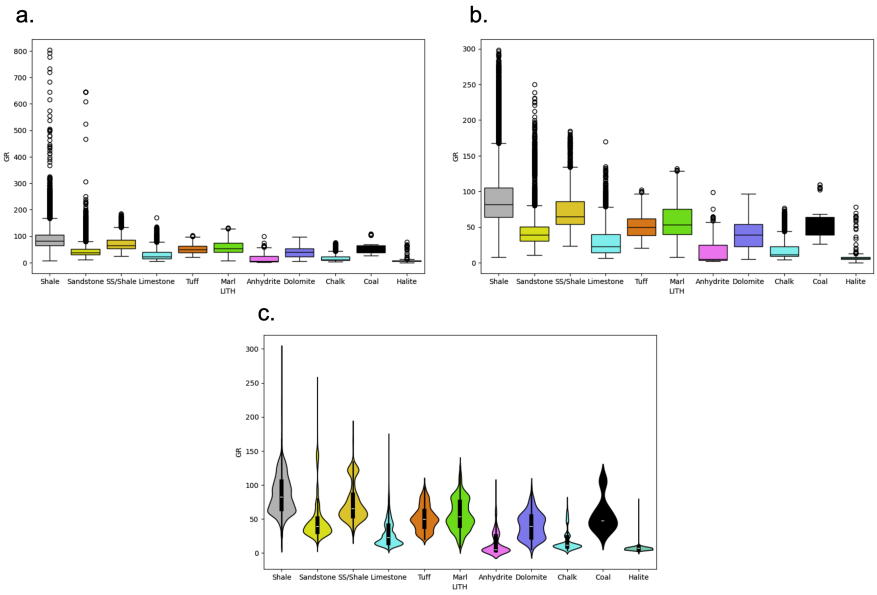


Figure 7.1: **a and b.** Box plots for the GR and lithologies in the wells. **c.** Violin plot for the same data.

As shown in Figure 7.1a, `GR` values exceeding 300 API are rare and likely represent spurious measurements. Let's remove these outliers from the dataset:

```
# remove GR values > 300 API
df_1 = df_1[df_1["GR"] < 300]

# GR in the different lithologies as box plot
fig = da.property_plot(df_1, "LITH", colors, "GR")
```

This code generates Figure 7.1b. We can also visualize the `GR` distribution using a violin plot. The following code generates this plot:

```
# GR distribution as violin plot
fig = da.property_plot(df_1, "LITH", colors, "GR", type="violin")
```

This code generates Figure 7.1c. The violin plot is a graphical representation that combines aspects of a box plot and a kernel density plot, showing the distribution of data, its density at different values, and its summary statistics, such as the median and quartiles.

7.2 Multivariate analysis

We now move on to multivariate analysis, where we examine relationships between multiple variables. To start, we'll explore the correlation between two variables using a cross plot—a simple yet powerful tool for identifying trends, patterns, and potential outliers.

The following function from our `analysis` module generates a cross plot of two variables and fits a linear trend line using SciPy's `linregress()` method, which performs a least-squares regression.

```
def cross_plot(df, col_1, col_2, class_col, colors, ax):
    """
    Cross plots two columns of the DataFrame df, and color the
    points by the classes in class_col.

    Input:
    df: DataFrame with the well logs
    col_1 and col_2: columns to cross plot
    class_col: string with the name of the column with the
        classes (e.g., "LITH" for lithology)
    cols: dictionary with the class names as the keys
        and the colors as [red, green, blue] values
    ax: axis to plot on

    Output:
    returns the slope, intercept and R2 of the line
    fit to the data
    """

    # create a scatter plot of the two columns
    sns.scatterplot(x=col_1, y=col_2, data=df,
                        hue=class_col, palette=colors, ax=ax)

    # create a dataframe with the two columns and drop NaNs
    df_s = df[[col_1, col_2]].dropna()
    # get the x and y data
    x, y = df_s[col_1], df_s[col_2]

    # fit a line to the data
    m, b, r, _, _ = linregress(x, y)
    # plot the line
    ax.plot(x, m * x + b, color="red", linestyle="--")

    ax.legend()

    return m, b, r**2
```

Let's use this function to cross plot neutron porosity (`AI`) versus density (`RHOB`) for all the wells, with the data points colored by lithology:

```
# cross plot NPHI vs RHOB
col_1, col_2 = "NPHI", "RHOB"

# create a figure and axis for the cross plot
fig, ax = plt.subplots()

# call the cross plot function
m, b, r_2 = da.cross_plot(df_1, col_1, col_2, "LITH", colors, ax)
ax.set_xlim(0, 1) # set x-axis limits (NPHI)
ax.set_ylim(1, 3) # set y-axis limits (RHOB)

# print the equation of the line and R2 value
print(f"{col_1} = {m:.2f} {col_2} + {b:.2f}, R2 = {r_2:.2f}")

plt.show()
```

This code produces Figure 7.2. We have now a predictive model for estimating neutron porosity from density and vice versa.

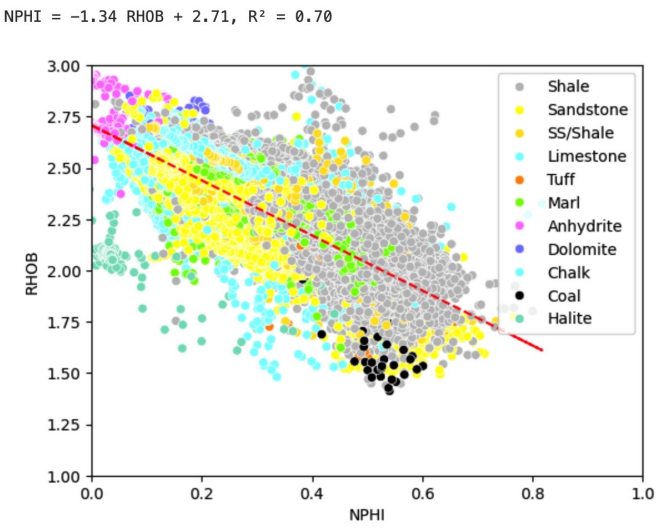


Figure 7.2: Cross plots of neutron porosity (NPHI) versus density (RHOB) for all the wells. Points are colored by lithology (LITH), and dash red line is best linear fit to the data.

Using the Seaborn `FacetGrid()` method, it is possible to map the dataset onto multiple axes arrayed in a grid of rows and columns that correspond to levels (or classes) of variables in the dataset. For example, to draw the same cross plot above, but separated by well name (`WELL`), we can do the following:

```
g = sns.FacetGrid(df_1, col='WELL', hue="LITH", palette=colors, col_wrap=4)
```

```
g.map(sns.scatterplot, 'NPHI', 'RHOB', marker="o", s=10, alpha=0.75)
g.set(xlim=(0, 1))
g.set(ylim=(1, 3))
g.add_legend(); # add semicolon to suppress the output
```

This code generates Figure 7.3, which illustrates the variation of different lithologies across the wells and highlights the correlation between `NPHI` and `RHOB` within each well.

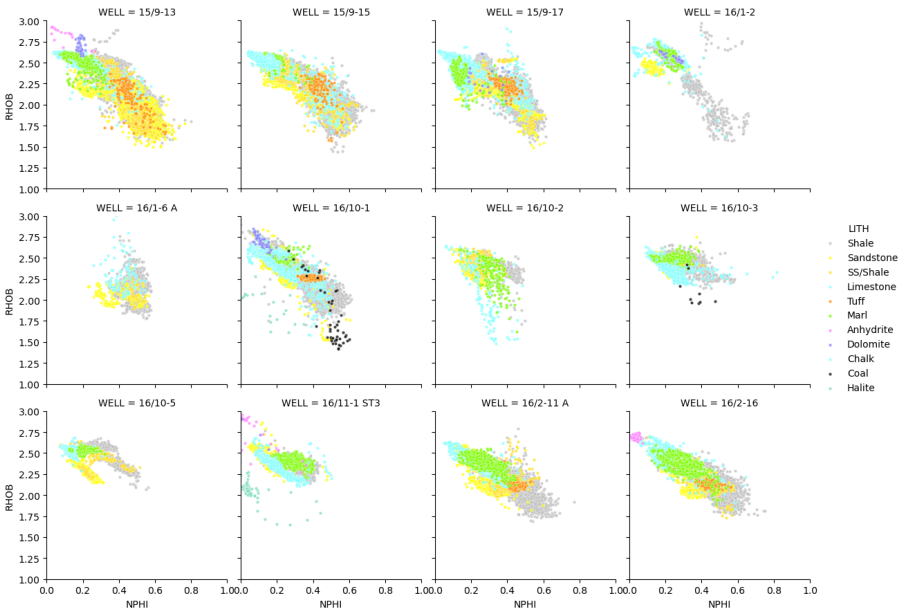


Figure 7.3: Cross plots of neutron porosity (NPHI) versus density (RHOB) for each well (WELL). Data points are colored by lithology.

Likewise, we can do the same plot with the data separated by groups (`GROUP`):

```
g = sns.FacetGrid(df_1, col='GROUP', hue="LITH", palette=colors, col_wrap=4)
g.map(sns.scatterplot, 'NPHI', 'RHOB', marker="o", s=10, alpha=0.75)
g.set(xlim=(0, 1))
g.set(ylim=(1, 3))
g.add_legend(); # add semicolon to suppress the output
```

This code generates Figure 7.3. An interesting group is the Upper Permian Zechstein, which exhibits significant lithological variability. Halite, being nearly incompressible, shows no correlation between `NPHI` and `RHOB`, whereas the other lithologies—except possibly shale—display a negative correlation between these two variables.

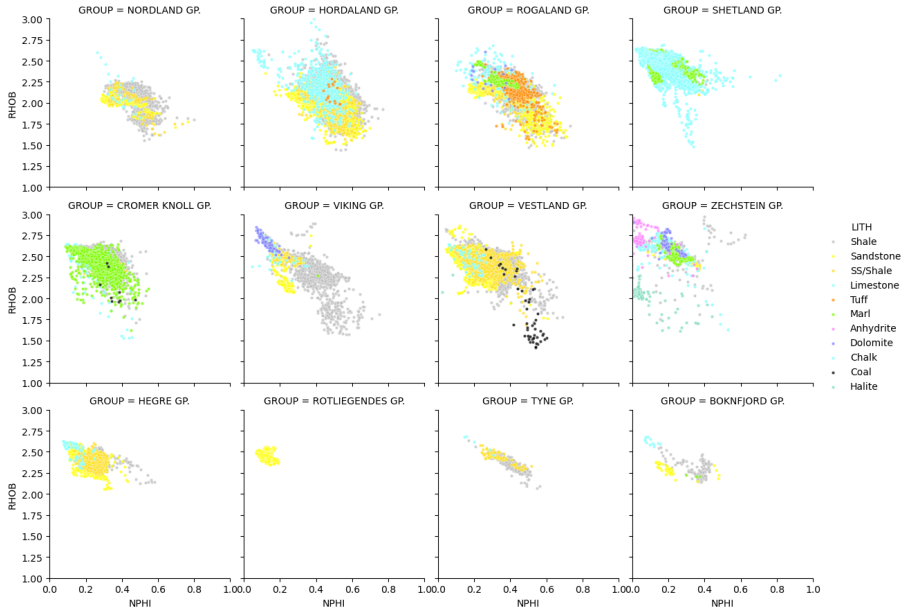


Figure 7.4: Cross plots of neutron porosity (NPHI) versus density (RHOB) for each group (GROUP). Data points are colored by lithology.

Using Seaborn's `pairplot()` method, it becomes straightforward to generate cross plots of multiple logs simultaneously. The code below cross plots four logs, with data points color-coded by lithology (`LITH`). Along the diagonal, since each variable cross plots against itself, a kernel density estimate (KDE) is shown (Figure 7.5).

```
# make a smaller DataFrame of the wells,
# with the logs of interest
df_3 = df_1[["GR", "RHOB", "NPHI", "DTC", "LITH"]]

# cross plot the logs and color points
# by LITH
sns.pairplot(df_3, hue="LITH",
                palette=colors,
                diag_kind="kde"); # ; suppress the output
```

The covariance and correlation matrices quantify the relationships between log variables. The covariance matrix measures how pairs of variables vary together—positive values indicate they increase or decrease together, negative values mean they move in opposite directions, and values near zero suggest little or no linear relationship. Diagonal elements show the variance of each variable. The correlation matrix standardizes these values, showing how strongly

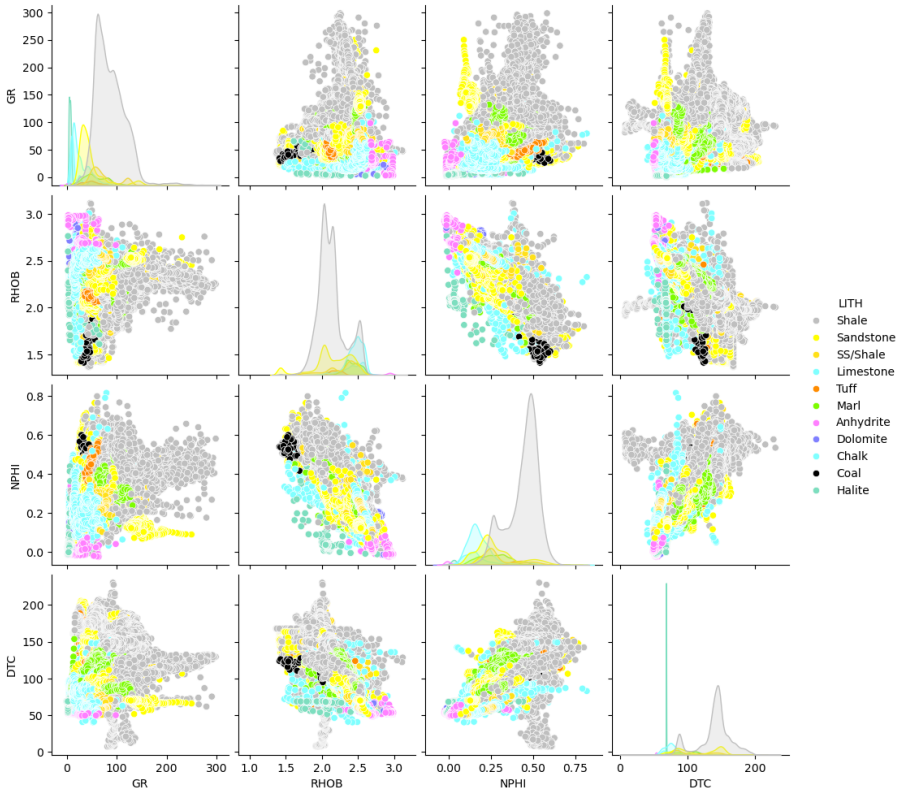


Figure 7.5: Cross plot of four selected logs in the wells' dataset. The data points are colored by lithology (LITH).

variables are linearly related, and whether they are positively correlated (both increase together) or negatively correlated (one increases as the other decreases). The code below generates these matrices for the log data in Figure 7.5. The result is shown in Figure 7.6.

```
# drop LITH column
if "LITH" in df_3.columns:
    # drop the unit name
    df_3 = df_3.drop(columns=["LITH"])

# plot covariance and correlation matrices
cov = df_3.cov()
corr = df_3.corr()

fig, axs = plt.subplots(1, 2, figsize=(10, 5))

# plot covariance matrix
sns.heatmap(cov, annot=True, fmt=".2f",
```

```
            cmap="coolwarm", vmin=-2,
            vmax=2, ax=axs[0])
axs[0].set_title("Covariance Matrix")

# plot correlation matrix
sns.heatmap(corr, annot=True, fmt=".2f",
            cmap="coolwarm", vmin=-1,
            vmax=1, ax=axs[1])
axs[1].set_title("Correlation Matrix")

fig.tight_layout()
plt.show()
```

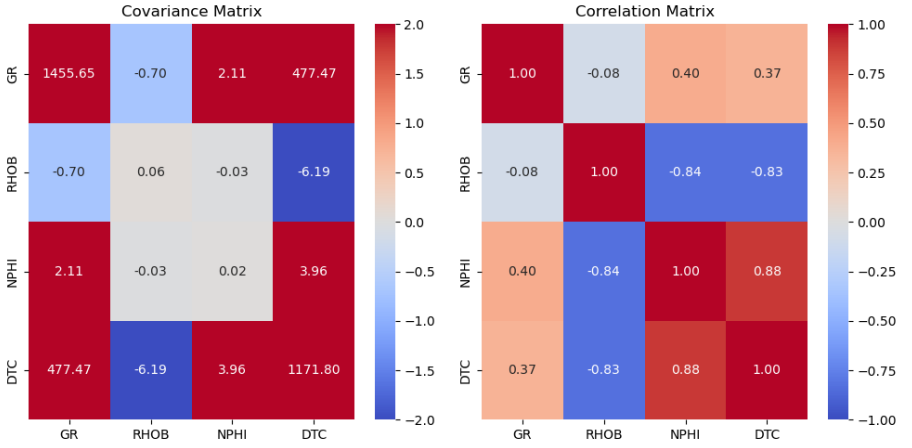


Figure 7.6: Covariance (left) and correlation (right) matrices for the logs in Figure 7.5.

7.3 Clustering data

In subsurface studies, it can be helpful to group similar log responses to identify patterns or zones with shared properties—especially when dealing with large datasets. This is where classification comes in. Instead of manually interpreting each log trace, we can use computational methods to automatically group similar data points.

One simple and widely used technique for this is K-Means clustering, a type of unsupervised learning. Unlike supervised methods, K-Means doesn't require labeled data—instead, it identifies clusters based solely on the structure of the data itself. The idea is to group data points so that those within the same group (or cluster) are more similar to each other than to those in other groups.

Thanks to the scikit-learn library in Python, performing K-Means clustering is both straightforward and efficient. The function below, from our `analysis` module, helps determine the optimal number of clusters for classifying the data—a crucial step, as selecting the appropriate number of clusters is often the most challenging part of the clustering process.

```
def optimal_clusters(data, max_k):
    """
    chooses the optimal number of clusters
    using the elbow and silhouette methods

    Input:
    data: DataFrame with the data to cluster
    max_k: maximum number of clusters to test

    Output:
    returns the figure
    """
    means_e = []
    means_s = []
    inertias = []
    scores = []

    for k in range(1,max_k+1):
        means_e.append(k)
        kmeans = KMeans(n_clusters=k, random_state=42)
        kmeans.fit(data)
        inertias.append(kmeans.inertia_)
        if k > 1:
            means_s.append(k)
            score = silhouette_score(data, kmeans.labels_)
            scores.append(score)

    fig, axs = plt.subplots(1,2,figsize=(10, 5))
    axs[0].plot(means_e, inertias, 'o-')
    axs[1].plot(means_s, scores, 'o-')

    y_labels = ["Inertia", "Silhouette Score"]
    titles = ["Elbow Curve", "Silhouette Curve"]
    for i, ax in enumerate(axs):
        ax.set_ylabel(y_labels[i])
        ax.set_title(titles[i])
        ax.set_xlabel("Number of Clusters")
        ax.grid()

    fig.tight_layout()
    plt.show()

    return fig
```

This next function, also from our `analysis` module, performs the actual clustering of the data using the K-Means algorithm.

```
def cluster_data(data, k):
    """
    Performs KMeans clustering

    Input:
    data: DataFrame (e.g., well logs) to cluster
    k: number of clusters to create

    Output:
    returns the KMeans object and the labels
    """
    # create the KMeans object
    kmeans = KMeans(n_clusters=k, random_state=42)
    # fit the model
    kmeans.fit(data)

    return kmeans, kmeans.labels_
```

In the example below, we apply K-Means clustering to four selected logs from well 16/10-1 to group the data into distinct clusters. We start by selecting the logs—`GR`, `RHOB`, `NPHI`, and `DTC`—then remove any missing values (a crucial step for the algorithm to function properly). Next, we use our helper function to determine the optimal number of clusters for the classification.

```
# well 16/10-1
df_well = df_1[df_1["WELL"] == "16/10-1"]

# key logs
df_4 = df_well[["DEPTH_MD", "GR", "RHOB", "NPHI", "DTC"]]

# drop NaNs
df_4 = df_4.dropna()

# data for clustering
data = df_4[["GR", "RHOB", "NPHI", "DTC"]]

# plot elbow curve
max_k = 10 # maximum number of clusters
fig = da.optimal_clusters(data, max_k)
```

This code produces an elbow plot and a silhouette score plot (Figure 7.7). Without going into details, these visualizations help estimate the optimal number of clusters: the elbow plot shows where adding more clusters no longer significantly reduces the within-cluster variance (typically at the "elbow" point), while the silhouette score plot indicates how well-separated the clusters are, with higher scores suggesting better-defined groups. Together, they guide us in selecting a number of clusters that balances simplicity with meaningful separation. For the selected log data, five clusters appear to provide a good balance.

The code below performs the clustering and generates cross plots of the logs, with data points color-coded according to their assigned cluster (Figure 7.8).

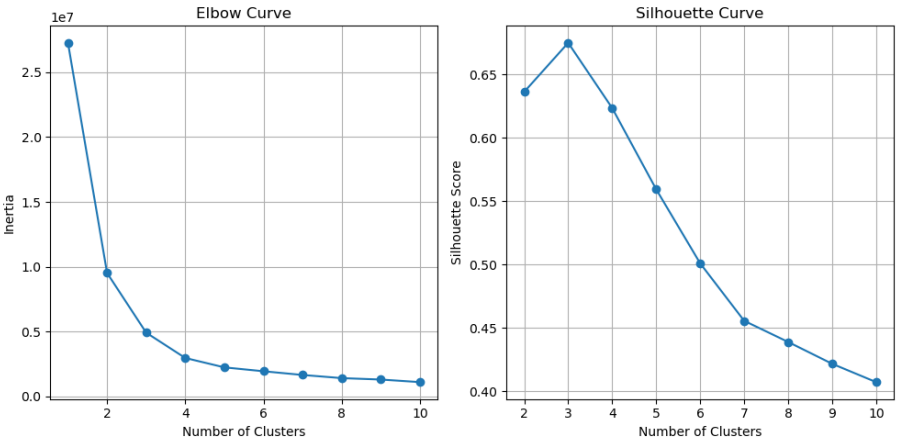


Figure 7.7: Elbow plot (left) and Silhouette scores (right) for KMeans classification of four logs from well 16/10-1.

```
# number of clusters
n_clusters = 5

# create a color palette for the clusters
palette = sns.color_palette("Set1", n_clusters)

# create a KMeans object
kmeans, labels = da.cluster_data(data, n_clusters)

# add the cluster labels to the DataFrame
df_4["cluster"] = labels

# plot the clusters
fig, axs = plt.subplots(1, 2, figsize=(10, 5))

# plot the clusters
sns.scatterplot(x="GR", y="RHOB",
                hue="cluster", data=df_4,
                palette=palette, ax=axs[0])
sns.scatterplot(x="NPHI", y="DTC",
                hue="cluster", data=df_4,
                palette=palette, ax=axs[1])

# set the titles
axs[0].set_title("GR vs RHOB")
axs[1].set_title("NPHI vs DTC")

plt.show()
```

Finally, we plot the logs alongside their corresponding cluster assignments as a function of depth (Figure 7.9).

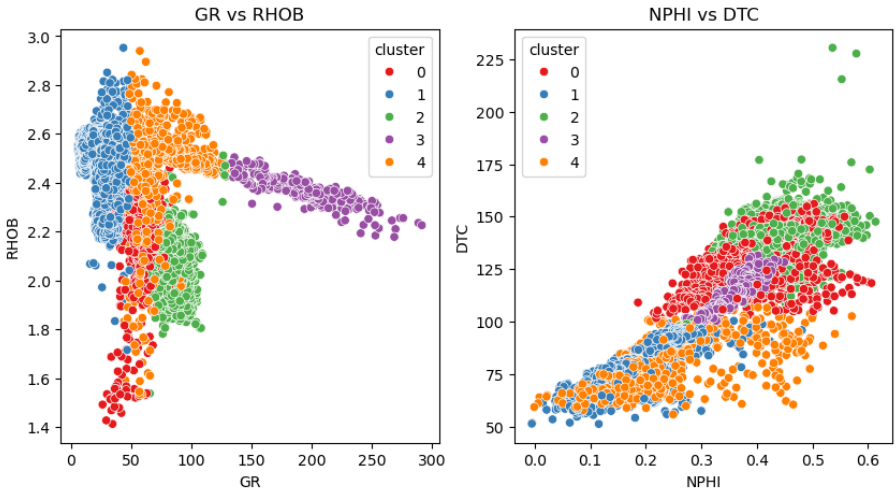


Figure 7.8: Cross plot of selected logs from well 16/10-1. The data points are colored by the cluster number.

```
# plot logs and clusters with depth
fig, axs = plt.subplots(1,5, figsize=(10, 10), sharey=True)

# label and invert the y axis
axs[0].set_ylabel("DEPTH (m)")
axs[0].invert_yaxis()

# plot logs
logs = ["GR", "RHOB", "NPHI", "DTC"]
for i, log in enumerate(logs):
    # plot the log as a blue curve
    axs[i].plot(df_4[log], df_4["DEPTH_MD"], '-', color="black", linewidth=0.5)
    # set the title
    axs[i].set_title(log)
    # set grid
    axs[i].grid()

# find the clusters tops
mask = df_4["cluster"] != df_4["cluster"].shift()
m_df = df_4[mask].reset_index(drop=True)

# plot the clusters
for i in range(len(m_df) - 1):
    # get the tops
    val1 = m_df.iloc[i]["DEPTH_MD"]
    val2 = m_df.iloc[i + 1]["DEPTH_MD"]
    # get the cluster
    cluster = m_df.iloc[i]["cluster"]
    # get the color
    color = palette[int(cluster)]
```

```
    # plot cluster as filled rectangle
    axs[-1].axhspan(val1, val2, facecolor=color)

axs[-1].set_title("Cluster")

fig.tight_layout()
plt.show()
```

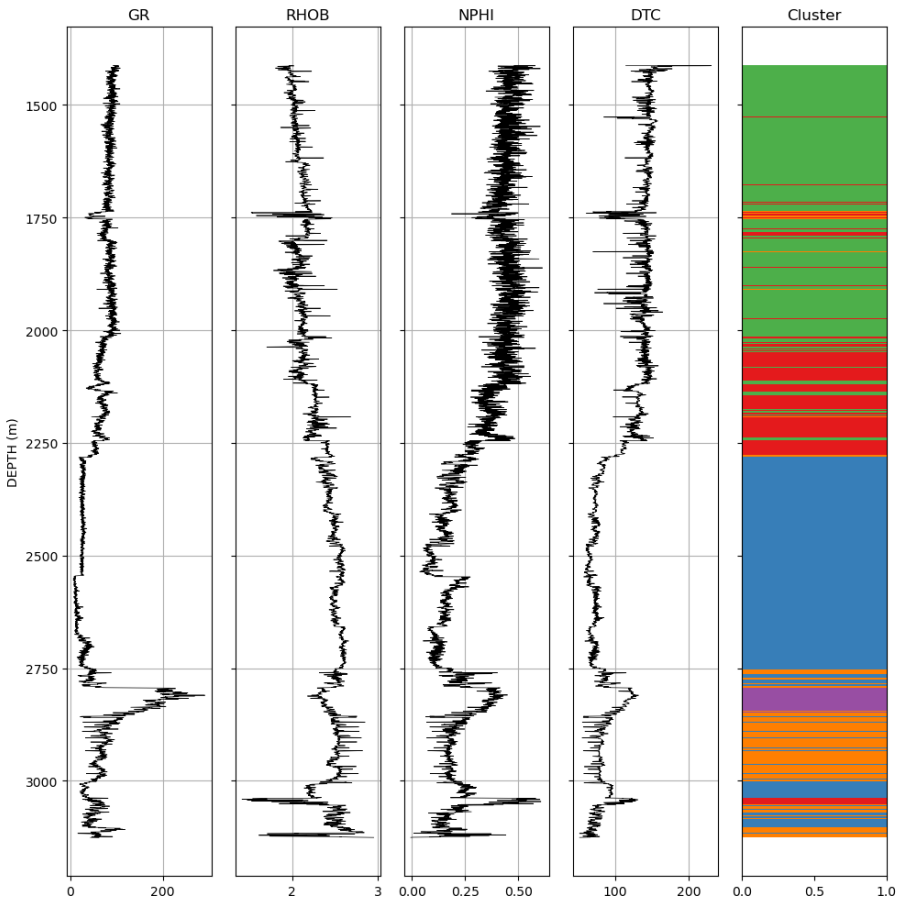


Figure 7.9: Selected logs and clusters in well 16/10-1.

The clusters in Figures 7.8 and 7.9 may correspond to different lithologies, but not necessarily. They could also reflect uncertainties, noise, or artifacts introduced during the classification process. It is important to corroborate these results with additional information—such as core data, facies logs, or geological context—before drawing definitive conclusions. As an example, you can compare the clusters log with the facies log in Figure 6.5.

7.4 Exercises

At this point, you're mature enough in your understanding to begin designing your own exercises. Consider exploring questions that interest you, experimenting with other datasets, or extending the methods introduced here to new scenarios. You might revisit some examples—such as analysing the properties of specific lithologies or groups across wells. The goal is to make the learning your own: ask questions, try things out, and don't be afraid to follow your inquiries.

Chapter 8

Advancing your skills

Congratulations on completing this course! You now have a solid foundation in Python—from basic data types and control flow to working with functions, classes, and modules. You've also explored how to visualize and analyze data using univariate and multivariate statistics. These are the essential tools that will support your future work in data-heavy, problem-driven environments like subsurface geosciences and engineering.

At this stage, you're ready to apply these skills to real-world projects. Whether that means using existing libraries or building your own tools, the key idea is this: Python is a means to an end. Your goal is not to write perfect code but to deliver insights, results, and solutions.

That said, how you write code still matters—especially if it needs to be used again, shared with a colleague, or maintained months down the line. While syntax is foundational, good structure is what makes code robust and reusable. Organizing your programs into functions, grouping related functionality into modules and packages, writing clear documentation, and testing your code will help others—and your future self—understand, trust, and build upon your work.

Some practical advice as you continue:

- **Start simple, then iterate.** Don't wait for the perfect design before writing code. Build something that works, then refactor it to make it clearer and more efficient. Tools like GitHub Copilot or ChatGPT can help you get started faster, suggest solutions, or explain unfamiliar code—but always review and test the suggestions critically.
- **Write tests for your code.** Even basic unit tests can help catch bugs early and make sure future changes don't break existing functionality. Python's unittest or pytest libraries are great places to start.

- **Lean on the community.** Stack Overflow, GitHub, and open-source libraries are full of examples, explanations, and solutions. These are some of my favorites: Awesome open geoscience, The Python Graph Gallery, Petrophysics Python Series, and xlines.
- **Keep learning.** Explore more specialized libraries for subsurface studies, such as lasio, welly, segyio, xarray, and geopandas. Or dive into machine learning with scikit-learn.
- **Remember your role.** You're not training to become a software engineer. You're a geoscientist or engineer using programming to solve complex, domain-specific problems. Delivering results matters most—but when done with well-structured, reusable, and tested code, your work will go further, last longer, and be more impactful. This paper is a great resource about that.