Implementierung von OM-Modellen und Verfahren in Python

Inka Nozinski, M.Sc., Timo Helfers, M.Sc.

January 24, 2025

Contents

Chapter 1

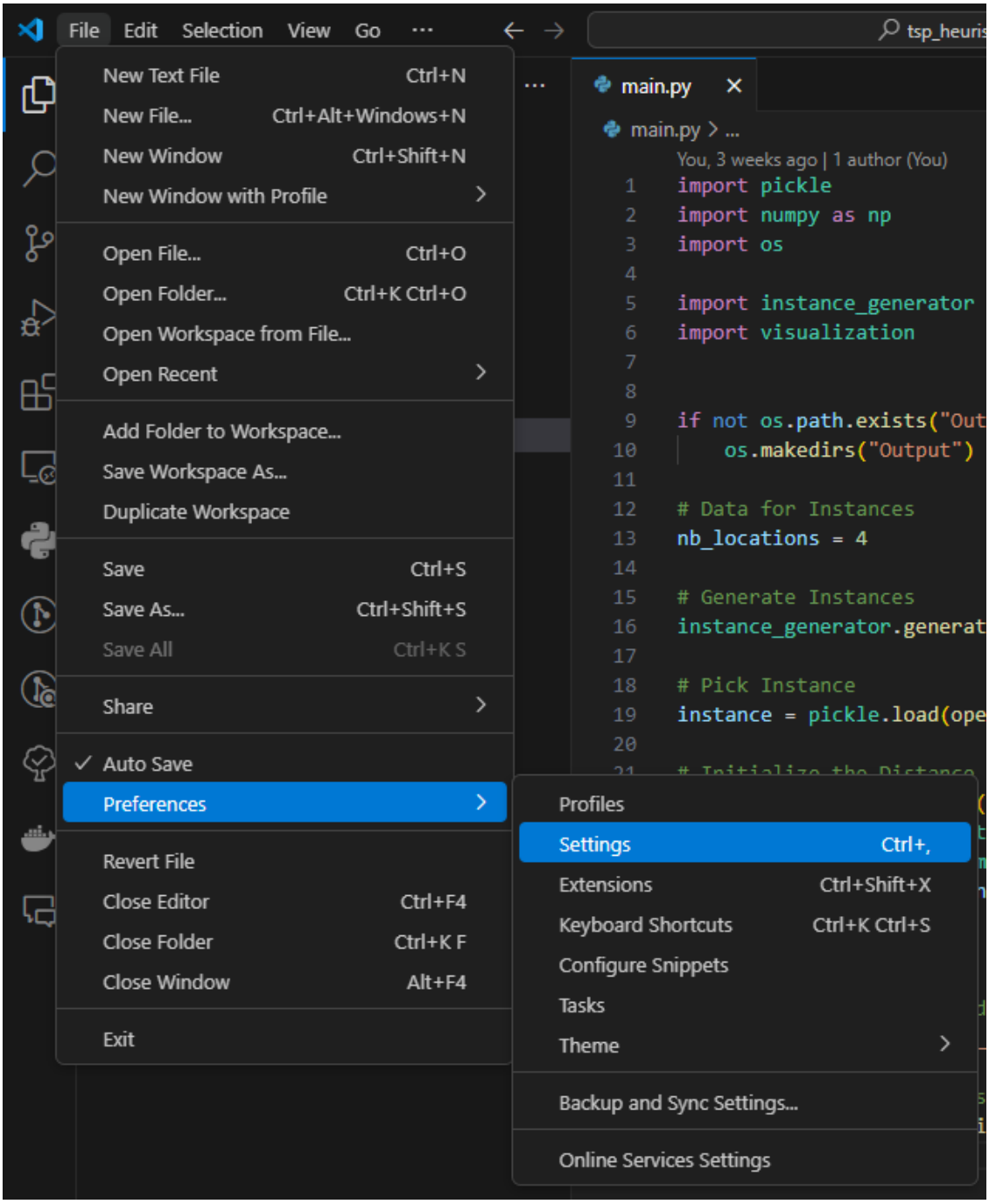
Getting started with VS Code

By now you should all have downloaded VS Code as described in the announcement in StudIP. VS Code is a source code editor developed by Microsoft that can be used on Windows, MacOS and Linux. VS Code is very popular with developers because it offers many features such as syntax highlighting, debugging, extensions and version control via git. You can also customise almost everything in VS Code. We will discuss some of this later.

With this notebook, we want to show you some commonly used settings and give you a quick overview of how to use VS Code.

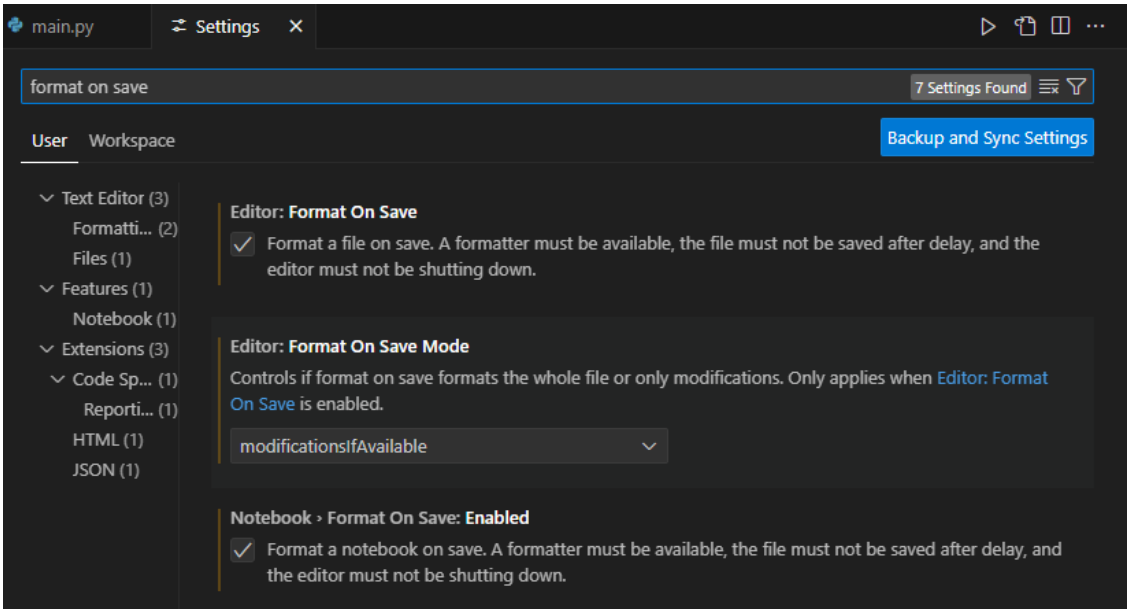
1.1 Open Settings

You can access your VS Code settings by clicking on `File` -> `Preferences` -> `Settings`. In VS Code you can nearly set everything as you like. In the following we look at two simple examples.



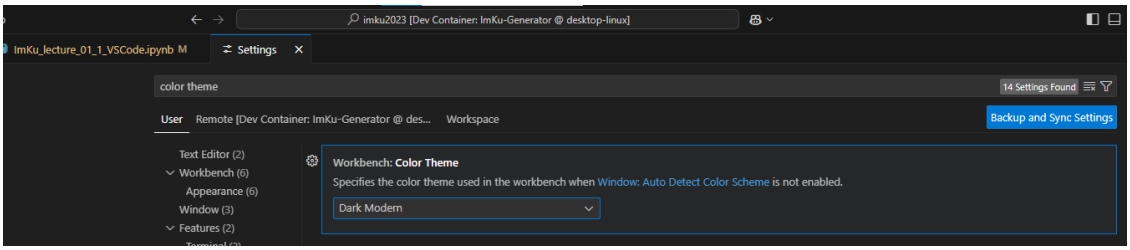
1.1.1 Autoformatting

Autoformatting is a feature in code editors that automatically formats your code to make it easier to read. It also formats the code according to the documentation of the programming language. For Python we use the Black Formatter. We recommend that you use this feature. Search in the settings for `format on save` and check the `Format on Save` box to format your document every time you save it. It will improve your code and makes it easier to follow for others. Using it the first time may take a while.



1.1.2 Color Theme

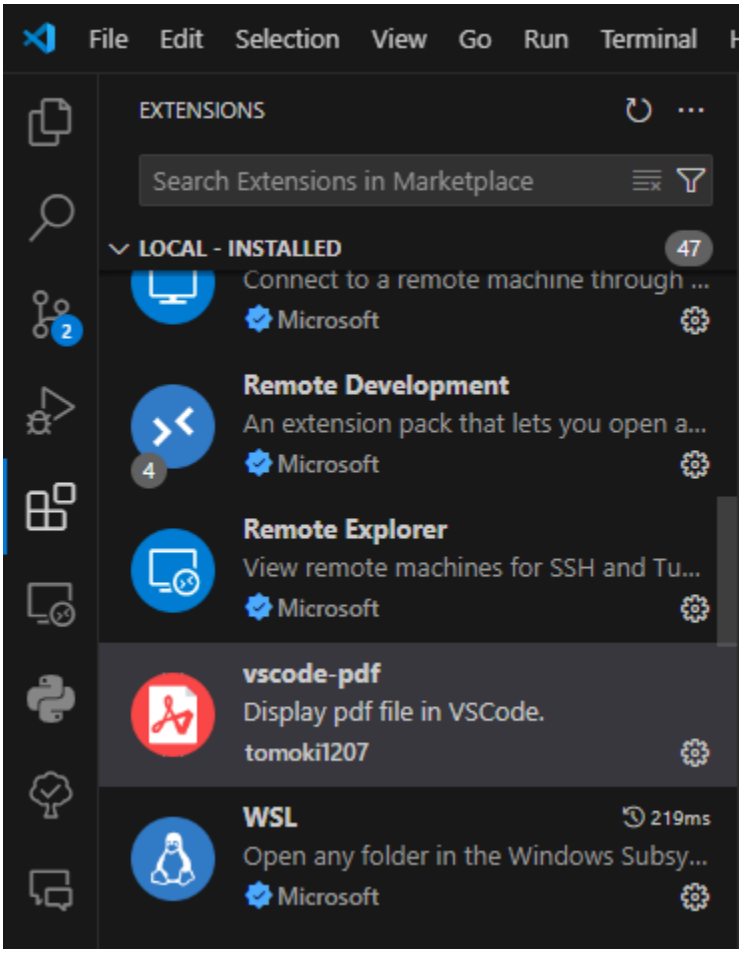
Another cool thing to set is the Color Theme. As you will spend much time looking at your screen choosing a color theme that feels pleasant for you is important. Depending on which color theme you choose the code will be highlighted differently, the background has a different color and the font sometimes differ. Choose a color theme by searching for `color theme` in settings.



When you scroll through the settings you can see the amount of things you can change and set. It's quiet a lot but for the beginning those two settings are the only things we change.

1.2 Extensions

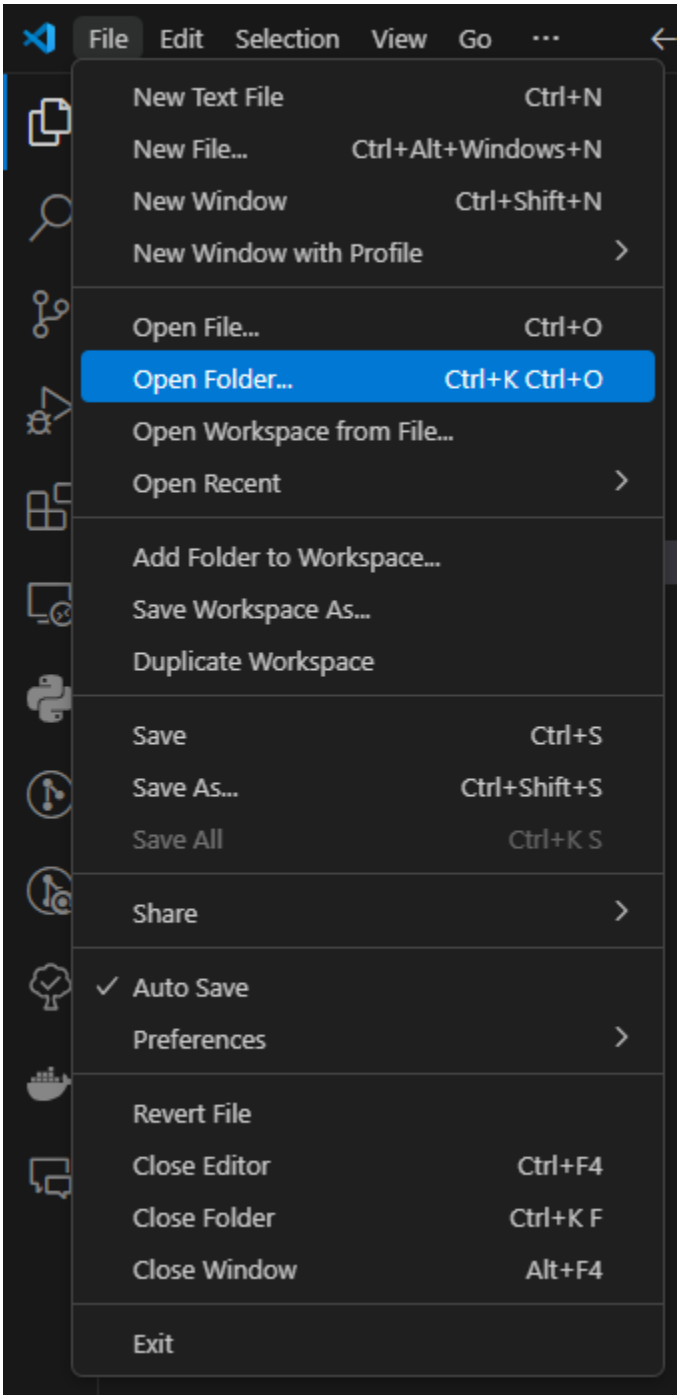
As previously mentioned there are many extensions ready to be installed in VS Code. Extensions are used to customise VS Code according to your needs. For example you can download an extension called `vscode-pdf` to be able to open pdf-files in VS-Code. This way you don't need to open another window with the pdf. There are many extensions to open different file types in VS Code. If you need a specific one just search for it in the extensions marketplace and try it.



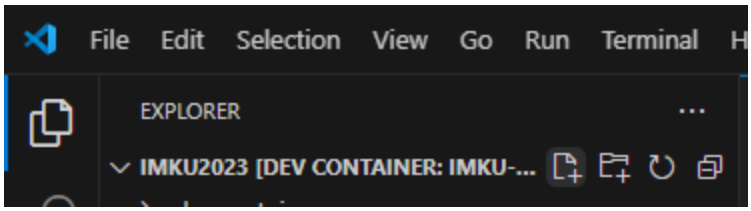
Other extensions help with your code. For example a spell checker or a different Autoformatter. Another cool feature is the extension `Todo Tree`. You can basically write a Todo-Comment wherever you want in your code, the extension finds all of them and places them on a list. It's not mandatory to use all kinds of extensions, it just helps you coding so don't be afraid to use them.

1.3 Open Folder

VS Code is for writing code, so how do we write? Firstly you should create a normal folder on your PC wherever you want to save it (for example your Desktop). After creating you can open your folder in VS Code by clicking on `File` -> `Open Folder...`.

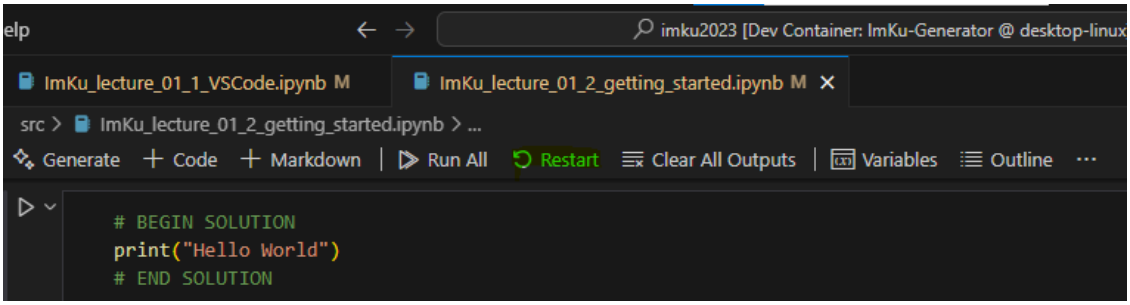


Once the folder is open, you can add files to it. Just click on the icon next to the name of your folder. Use a short name for your file and write what type of file it should be. If the file should be a notebook write `.ipynb`. If it's a normal python script write `.py`. You can work on almost any file type with VS Code by using the correct extension.



1.4 Python Module in Jupyter Notebook

When working with Python you will almost always import modules in which you have implemented another function. This is not a problem in jupyter notebooks, but remember to restart your kernel every time you change something in your modules. The changes will only be synchronised after the restart is done.



1.5 Some nice Shortcuts

Here you can find some nice shortcuts. These can save you a lot of time when coding. Don't worry, you don't have to use them. They just come in handy sometimes.

1.5.1 Editor Shortcuts

Name	Operator
Strg + S	Save
Strg + Z	Undo
Strg + S	Save
Strg + C	Copy
Strg + V	Paste
Strg + F	Search
Strg + H	Replace
Strg + #	Comment selected lines out
F2	Rename all occurrences
F8	Go to next error/warning
Strg + .	Open Quickfix-Menu

1.5.2 Navigation Shortcuts

Name	Operator
Strg + Arrow left/right	navigate through words
Strg + Arrow up/down	navigate through lines

Name	Operator
Pos1	go to beginning of line
End	go to end of line

1.5.3 Code Manipulation Shortcuts

Name	Operator
Alt + Arrow up/down	Moves selected lines up/down
Alt + Shift + Arrow up	Duplicate selected line
Strg + Space	Open IntelliSense
Strg + D	Edit multiple occurrences
Strg + U	Unselect last occurrence
Strg + Shift + L	Select all occurrences
Strg + Alt + Arrow up	Add multicursor

Chapter 2

Getting Started with Python

2.1 My first Program: `Hello World!`

The first code you write in a new programming language usually outputs the following text: `Hello World`. This is just a single line of code in Python. In this case, print does not mean to literally print something out, but to display the text on the standard output, meaning the screen.

```
In [1]: print("Hello World")
```

```
Hello World
```

2.2 Using Python as a Calculator

In some sense, Python is like a calculator on steroids. It can do most things that you are used to from normal calculators (like basic arithmetic). We can print the solution in the same way as with `print('Hello World')`. Let's calculate 2+2.

```
In [2]: print(2 + 2)
```

```
4
```

In a Jupyter-Notebook, it is also possible to do this without the print()-Statement.

```
In [3]: 2 + 2
```

```
Out[3]: 4
```

Let's calculate the area of a circle ($\pi \cdot r \cdot r$) with radius of $r = 2$. Use 3.14159 for π. *Note: Later we will learn exponentiation in Python. For now, however, it also works like this.*

```
In [4]: print(2 * 2 * 3.14159)
```

```
12.56636
```

Without exponentiation, we have to write the radius two times, which is a bit annoying. To make this easier, we can store the radius in a variable. This is done by using the assignment operator `=`. We can also store the value of pi in a variable, so we can reuse it later. When we write multiple lines, the lines will be executed in order, and the result of the last line will be displayed. Let's use a radius $r = 3$. *Note: For things line π, there are libraries in Python that can be used. We will learn more about this later.*

```
In [5]: r = 3
        pi = 3.14159
        print(r * r * pi)
```

```
28.27431
```

Variables can also be changed during runtime, they are called 'variables' after all.
Print the variable r, than change the variable r to the value 2 and print it again.

```
In [6]: print(r)
        r = 2
        print(r)
```

```
3
2
```

If you want to print two statements in one cell, you have to use print statements, as only the last statement will be displayed. It therefore makes sense to **always** use `print()` in order to avoid mistakes.

```
In [7]: r  # this r will not be displayed
        r = 4
        r
```

```
Out[7]: 4
```

With r and pi stored in variables, we can also calculate the circumference of the circle $(2 \cdot r \cdot \pi)$.

```
In [ ]: print(2 * r * pi)
```

```
25.13272
```

We can also store the area in a variable. In general, everywhere where you can write a value, you can also write an expression. This is what makes programming so powerful.

```
In [9]: area = r * r * pi
        print(area)
```

```
50.26544
```

In some cases, we are not interested in too many decimal points. We can use `round()` in those cases. We can either round the variable directly or only round the displayed value in the `print()` statement.

```
In [10]: print(area)  # original value
         print(round(area, 2))  # rounded in the print statement
         print(area)  # did not change the original value
         area = round(area, 2)  # changed the original value
         print(area)  # changed value
```

```
50.26544
50.27
50.26544
50.27
```

Strings are also a kind of value, but they are not numbers, so we can't do arithmetic with them. We can, however, concatenate them using the `+` operator. Combine the words `Hello` and `World` to one string that is stored in a variable.

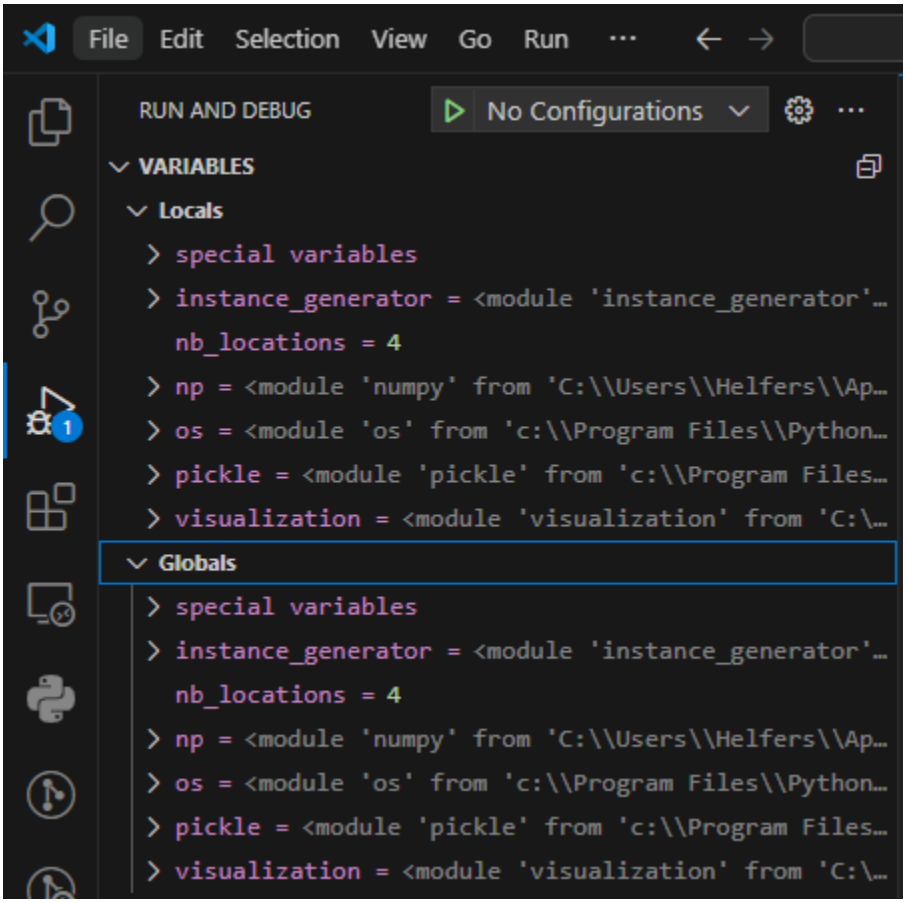
```
In [11]: first_word = "Hello "
         second_word = "World"
         combined_string = first_word + second_word
         print(combined_string)
```

```
Hello World
```

However, you cannot add different types with the `+` operator. Both sides of the `+` operator must be strings to concatenate or numbers to add, so the following code does not work.

```
In [12]: # print('circle area = ' + area) # uncomment this line to see the error
```

Strings are always placed in quotation marks. Both single `'` and double `"` quotation marks can be used. For the sake of clarity, however, a constant change in the type of quotation marks should be avoided.



If quotation marks are to be used within a string, we use the respective other quotation mark.

```
In [13]: print('This is a "string" with a single quote.')
         print("This is a 'string' with double quotes.")
```

```
This is a "string" with a single quote.
This is a 'string' with double quotes.
```

2.3 Basic Data Structures

We can check the type of a value using the `type()` function.

```
In [14]: type(area)
```

```
Out[14]: float
```

The four most important basic data types in Python are:

- `int` for integers or whole numbers
- `float` for floating point numbers or real numbers
- `str` for strings meaning sequences of characters
- `bool` for booleans or logical values (more on this later)

Every value or variable has a type. Since Python is a dynamically typed language, the type of a variable is determined automatically, and we mostly don't see it. It is still there, however.

```
In [15]: print(type(2))
         print(type(2.5))
         print(type("Hello World"))
         print(type(True))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

2.4 More on `print()` Statements

As we have already learned, we can output variables, e.g. the radius, the area and π.

```
In [16]: print(r)
         print(area)
         print(pi)
```

```
4
50.27
3.14159
```

However, the output of pure numbers quickly becomes confusing. It would be better to add additional information in the `print()` statement. Thus, we need to print different data structures within one `print()` statement. There are different ways to do so.

1. We can convert a number, e.g. a float to a string with `str()` and add two strings with `+` as above.

```
In [17]: print("For r = " + str(r) + ", the area is " + str(area) + ".")
```

```
For r = 4, the area is 50.27.
```

2. We can also print multiple arguments separated by a comma. Those arguments can have different types.

```
In [18]: print("For r =", r, ", the area is ", area, ".")
```

```
For r = 4 , the area is  50.27 .
```

3. We can use f-strings. Within the `f-string`, the variables can be output in curly brackets. **We recommend this variant** as it enables a very compact notation.

```
In [19]: print(f"For r = {r}, the area is {area}.")
```

```
For r = 4, the area is 50.27.
```

If we want to make a comment in a cell, we can do this with `#`.

```
In [20]: # this is a comment
         print("Hello!")  # This is also a comment

         # print("Hello!") # We can also comment out code
```

```
Hello!
```

2.5 Control Structures

2.5.1 `if` and `else`

What makes programming powerful is the power to do *something*, *something else* or even *nothing at all*. For this, we use the `if else` construct: In this example, we want to print something different depending on the value of `my_number`. The statements for `if` and `else` are grouped by indentation. The equal sign is of great importance. One equals sign `=` means an assignment, whereas two equal signs `==` mean a check.

```
In [21]: my_number = 2  # assignment

         if my_number == 2:  # check for equality
             print("Your number is 2.")
         else:
             print("your number is not 2.")
```

```
Your number is 2.
```

Let's check whether the number is smaller, larger or equal to two. We use `elif` in this case.

```
In [22]: my_number = 4
         if my_number == 2:  # check for equality
             print("Your number is 2.")
         elif my_number > 2:  # check for greater than 2
             print("Your number is greater than 2.")
         else:
             print("Your number is not greater than 2.")
```

```
Your number is greater than 2.
```

2.5.2 `for` Loops

Automating tasks often involves performing an action multiple times. To achieve this, we use loops. There are two main types of loops in Python: `while` loops and `for` loops. As before, the statements within a loop are grouped by indentation. Let's print `Hello World` ten times. We can use `range()` for this, which provides us with a sequence of numbers.

```
In [23]: print(list(range(10)))
```

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

```
In [24]: for i in range(10):
             print("Hello World")
             # print(i, 'Hello World') # uncomment this line to see the numbers
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

If we want to print the numbers from 1 to 10, we need to define the start and end of `range()`. The last number is not included in `range()`.

```
In [25]: for i in range(1, 11):
             print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Let's combine what we learned above: Compute and print the are of a circle with a range of 1, 2, 3, 4 and 5 using a `for` loop and a `f-string`.

```
In [26]: for i in range(1, 6):
             area = i * i * pi
             print(f"For r = {i} cm, the area is {area} cm^2.")
```

```
For r = 1 cm, the area is 3.14159 cm^2.
For r = 2 cm, the area is 12.56636 cm^2.
For r = 3 cm, the area is 28.27431 cm^2.
For r = 4 cm, the area is 50.26544 cm^2.
For r = 5 cm, the area is 78.53975 cm^2.
```

You can use `continue` to skip the rest of the current iteration and continue with the next one. This is useful if you want to skip some values in a loop. Let's compute the sum of all even numbers between 1 and 10 using `continue`.

Note: This teaching example is intended to help you understand the function of ***`continue`****. There are better ways to calculate the sum of the even numbers.*

```
In [27]: sum_even_numbers = 0

         for number in range(1, 11):
             # imagine there are a bunch of reasons not to do the complicated thing
             if number % 2 != 0:
                 # skip odd numbers
                 continue

             # imagine that this part of the code is very complicated
             print(number)
             sum_even_numbers += number

         print(f"The result is {sum_even_numbers}.")
```

```
2
4
6
8
10
The result is 30.
```

Using `continue` can make your code more readable since you sort out all the exceptional cases first and then do the actual work. In this case, our exception is if the number is not even. Another benefit of this is that it reduces indentation by one. Otherwise, everything would be indented by one more level inside the `if` statement.

You can use `break` to exit a loop early. This is useful if you are searching for something. Once you have found it, it doesn't make sense to keep on searching, so it makes sense to exit the loop early.

Let's use `break` to you find the number 3 in the numbers from 1 to 10.

```
In [28]: number = 3
         for i in range(1, 11):  # numbers 1 to 10
             print(f"The number is {i}.")
             if i == number:
                 print("Found the number")
                 break
```

```
The number is 1.
The number is 2.
The number is 3.
Found the number
```

2.5.3 `while` Loops

The above can also be done with slightly more effort using a `while` loop: A while loop runs as long as the condition is `True`. Print the numbers from 1 to 10 using a `while` loop.

```
In [29]: counter = 1
         while counter <= 10:
             print(counter)
             counter = counter + 1
```

```
1
2
3
4
5
6
7
8
9
10
```

`While` and `for` loops can also be combined. Output the numbers from 1 to 5 exactly three times in succession using a `for` loop with `range` within a `while` loop.

```
In [30]: repetition = 1

         while repetition <= 3:
             print(f"Repetition: {repetition}")

             # inner loop for the numbers
             for i in range(1, 6):
                 print(i)

             # increase the counter by one
             repetition += 1
```

```
Repetition: 1
1
2
3
4
5
Repetition: 2
1
2
3
4
5
Repetition: 3
1
2
3
4
5
```

2.6 Operators

2.6.1 Mathematical Operators

Name	Operator	Example
Addition	+	2 + 2 = 4
Subtraction	-	5 - 3 = 2
Multiplication	*	2 * 3 = 6
Division	/	7 / 4 = 1.75
Integer Division	//	7 // 4 = 1
Modulo	%	7 % 4 = 3
Exponentiation	**	2 ** 3 = 8

Use division and integer division.

```
In [31]: print(f"Division: {7/5}")
         print(f"Integer Division: {7//5}")
```

```
Division: 1.4
Integer Division: 1
```

Write a statement that prints `True` if the modulo of $\frac{8}{5}$ is less than two and `False` otherwise.

```
In [32]: mod = 8 % 5
         if mod < 2:
             print(True)
         else:
             print(f"{False}, the modulo is {mod}.")
```

```
False, the modulo is 3.
```

2.6.2 Relational Operators

The result of relational operators is always either `True` or `False` this is called a boolean value.

Name	Operator	Example
Equal	==	2 == 2 -> True
Not Equal	!=	2 != 2 -> False
Greater	>	2 > 3 -> False
Greater or Equal	>=	3 >= 2 -> True
Less Than	<	4 < 4 -> False
Less Than or Equal	<=	4 <= 4 -> True

Note the difference between comparison `==` and assignment `=`: `==` checks for equality of the left and right side, while `=` assigns the right side to the left side.

2.6.3 Logical Operators

Lastly, there are the logical operators `and`, `or` and `not` These are used to combine boolean values. The effect of these can be seen in the so-called *truth tables*.

x	y	x and y	x or y	not x
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Use the `and` operator and an `if` condition to check whether the person is allowed to drive.

```
In [33]: age = 20
         driving_license = True

         if age >= 18 and driving_license:
             print("Allowed to drive.")
         else:
             print("Not allowed to drive.")
```

```
Allowed to drive.
```

Check if at least one of the numbers is equal to two.

```
In [34]: x = 2
         y = 3

         if x == 2 or y == 2:
             print("At least one of the numbers is 2.")
```

```
At least one of the numbers is 2.
```

2.7 Basic Functions

Very often, we want to do the same thing in different places in our code. Instead of writing it all over again, we should put the code into a function. Functions are similar to mathematical functions, but they can do a lot more.

We can declare a function with `def name():`. Let's create a function that gives use the square of a number.

```
In [35]: # create the function
         def square(x):
             return x * x
```

```
In [36]: # call the function with different arguments
         s = square(2)
         print(s)
         print(square(3))
```

```
4
9
```

A function can also have multiple arguments. Create a function returning the maximum of two numbers.

```
In [37]: def maximum(a, b):
             if a > b:
                 return a
             else:
                 return b


         print(maximum(2, 3))
         print(maximum(6, 5))
```

```
3
6
```

Functions can do much more, like actually doing something besides returning a value. For example, they can print a value.

Create a function that print `hello there`.

```
In [38]: def print_greeting():
             print("hello there")  # we have no return in this function


         print_greeting()
```

```
hello there
```

Write a function that prints whether a given number is divisible by 3 without a remainder.

```
In [39]: def divisible_by_three(n):
             if n % 3 == 0:
                 print(f"{n} is divisible by 3.")
             else:
                 print(f"{n} is not divisible by 3.")


         divisible_by_three(5)
         divisible_by_three(6)
```

```
5 is not divisible by 3.
6 is divisible by 3.
```

2.8 Examples

2.8.1 FizzBuzz

FizzBuzz is a simple programming exercise that is based on a children's game. The rules are as follows:

- Count up through the whole numbers (from 1 to 30 in our case).
- If a number is divisible by 3 print Fizz instead of the number.
- If a number is divisible by 5 print Buzz instead of the number.
- If it is divisible by both 3 and 5 print FizzBuzz.

```
In [40]: for i in range(1, 30 + 1):
             if i % 3 == 0 and i % 5 == 0:
                 print("FizzBuzz")
             elif i % 3 == 0:
                 print("Fizz")
             elif i % 5 == 0:
                 print("Buzz")
             else:
                 print(i)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
```

2.8.2 Factorial

Write a function to implement the Factorial **!** operator.

The factorial of a natural number is defined as follows:

$$n! := \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdots n$$

```
In [41]: def factorial(n):
             value = 1  # start with 1 as 0! = 1
             for i in range(1, n + 1):
                 value = value * i
                 # value *= i # this is the same as the line above
             return value


         print(factorial(0))
         print(factorial(1))
         print(factorial(2))
         print(factorial(3))
```

```
1
1
2
6
```

2.8.3 Optimal Order Quantities

Economic Order Quantity (EOQ) - formula helps businesses find the optimal order quantity (q^*) for a product with a constant demand rate. It minimizes total inventory costs by balancing two factors:

Ordering Cost (s): The cost to place an order. Holding Cost (h): The cost to hold one unit in inventory for a year.

The EOQ formula is:

$$q^* = \sqrt{\frac{2 \cdot s \cdot \tilde{d}}{h}}$$

Where:

- q^* = Economic Order Quantity
- $\tilde{d}$ = Annual demand (constant)
- s = Ordering cost per order
- h = Holding cost per unit per year

By using this formula, businesses can minimize inventory costs while maintaining adequate stock levels to meet demand.

```
In [42]: ordering_cost = 200
         annual_demand = 1000
         holding_cost = 10


         def calculate_EOQ(s, d, h):
             return ((2 * s * d) / h) ** 0.5
```

```
        eoq_result = calculate_EOQ(ordering_cost, annual_demand, holding_cost)
        print(f"The Economic Order Quantity (EOQ) is: {eoq_result}")
```

```
The Economic Order Quantity (EOQ) is: 200.0
```

Chapter 3

Data Structures

3.1 Lists

3.1.1 Basics

If we want to store multiple elements in one place, we can use a list. Lists are also types, or more accurately, compound types, because they are made up of other types. This is opposed to the atomic types like `int`, `float`, `str` and `bool` that we have seen so far. Similar to lists in real life, a list is an ordered collection of *things*. A list is created with square brackets. Let's create two lists with numbers.

```
In [1]: list1 = [1, 2, 3]
        list2 = [4, 5, 6]
```

A list can also contain other data types, e.g. strings, or even multiple data types. We can also have an empty list.

```
In [2]: list3 = ["Tim", "John", "Alice"]
        list4 = [42, "hello", "world", list1, [4, 5, 6]]
        empty_list = []
```

Just as for other data types, we can use the `print()` and `type()` statement with lists.

```
In [3]: print(list1)
        print(type(list3))
```

```
[1, 2, 3]
<class 'list'>
```

3.1.2 Accessing Elements - Slicing Lists

We can access the individual elements in a list using the `[]` operator and the index of the element we want to access. Note that the indices start at 0, so if we want to get the second element of a list, we have to use the index 1.

```
In [4]: my_list = ["First", "Second", "Third", "Fourth"]

        print(my_list[1])  # Element at index 1 (not the first element)
        print(my_list[0])  # Element at index 0 (the first element)
```

```
Second
First
```

The elements of a list are counted from the end when using negative indices, e.g. `[-1]`.

```
In [5]: print(my_list[-1])  # last element
        print(my_list[-3])  # third last element
```

```
Fourth
Second
```

In addition to single-element access, Python also offers a slicing operator `[:]` to easily get slices of a list. The operator takes in two arguments: the start and end index of the slice. The start index is inclusive, and the end index is exclusive. Both arguments are optional and default to the start and end of the list, respectively.

Get the second, third and fourth element from the list.

```
In [6]: other_list = ["I", "like", "lists", "in", "python", "!"]

        print(other_list[2:5])  # starting index is inclusive, the ending index is␣
↪exclusive
```

```
['lists', 'in', 'python']
```

If the start or end is not specified, it starts or ends with the list.

```
In [7]: print(other_list[:3])  # start with list
        print(other_list[4:])  # end with list
```

```
['I', 'like', 'lists']
['python', '!']
```

If negative indices are used, the last elements of the list are omitted.

```
In [8]: print(other_list[:-1])
        print(other_list[:-3])  # leaves out the last two elements
```

```
['I', 'like', 'lists', 'in', 'python']
['I', 'like', 'lists']
```

There is also a variation on the slicing operator with two colons `[::]` that lets you specify a step size. Print every second element starting at index 1 end ending a index 5.

```
In [9]: print(other_list[1:5:2])
```

```
['like', 'in']
```

A negative step size can also be used to reverse a list.

```
In [10]: print(other_list[::-1])
```

```
['!', 'python', 'in', 'lists', 'like', 'I']
```

3.1.3 Changing Lists

Lists can be concatenated using the + operator and added repeatedly using *.

```
In [11]: list1 = [1, 2, 3]
         list2 = [4, 5, 6]
         new_list_1 = list1 + list2
         print(new_list_1)
         new_list_2 = list1 * 5
         print(new_list_2)
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

We can add elements to an existing list using the `append()`.

```
In [12]: my_list = [2, 4, 6]
         print(my_list)

         my_list.append(1)  # add an element to the end of the list
         print(my_list)
```

```
[2, 4, 6]
[2, 4, 6, 1]
```

We can also start with an empty list and add elements. Let's add the numbers from 1 to 10.

```
In [13]: list_of_numbers = []

         for i in range(1, 11):
             list_of_numbers.append(i)  # append i to the list
             print(list_of_numbers)
```

```
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
```

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

We can add an element at a specific position with `insert()`. The first value is the new index of the element and the second value is the element itself.

Let's add the number 12 at the second position.

```
In [14]: my_list = [2, 4, 6]
         my_list.insert(1, 12)  # (new index, value)
         print(my_list)
```

```
[2, 12, 4, 6]
```

With `pop()`, we can get **and** remove the last element of the list.

```
In [15]: print(my_list)
         print(my_list.pop())
         print(my_list)
```

```
[2, 12, 4, 6]
6
[2, 12, 4]
```

Let's print and remove each element from the list using a `while` loop.

```
In [16]: my_list = [2, 8, 4, 7]

         while my_list:  # an empty list evaluates to False
             x = my_list.pop()
             print(f"x = {x}, my_list contains {my_list}")
```

```
x = 7, my_list contains [2, 8, 4]
x = 4, my_list contains [2, 8]
x = 8, my_list contains [2]
x = 2, my_list contains []
```

With `pop()`, we can also get and remove an element by an index. Let's get and remove the third element of the list.

```
In [17]: my_list = [2, 8, 4, 7]
         print(my_list.pop(2))
         print(my_list)
```

```
4
[2, 8, 7]
```

With `sort()`, we can sort a list. This works with list of strings and list of numbers (but not with mixed lists).

```
In [18]: string_list = ["g", "r", "a"]
         int_list = [5, 1, 8]
         float_list = [1.3, 1, -1.6]
         mixed_list = ["A", 0.8, 5]

         # sort a list
         string_list.sort()
         int_list.sort()
         float_list.sort()
         # mixed_list.sort() # not possible, uncomment to see the error

         print(string_list)
         print(int_list)
         print(float_list)
```

```
['a', 'g', 'r']
[1, 5, 8]
[-1.6, 1, 1.3]
```

A list can be reversed with `reverse()`.

```
In [19]: my_list = [1, 2, 3, 4, 5]
         my_list.reverse()
         print(my_list)
```

```
[5, 4, 3, 2, 1]
```

3.1.4 Special Functions for Lists

Often, the length of a list is required. It is determined with `len()`.

```
In [20]: print(len(list1))
         print(len(empty_list))
```

```
3
0
```

With `in`, it can be checked if a value is in a list. The returned value is a boolean (`True` or `False`).

```
In [21]: my_list = ["a", "b", "c", "d", "e"]

         print("b" in my_list)
```

```
True
```

`min()` and `max()` return the minimum and maximum value of a list.

```
In [22]: my_list = [1, 2, 3, 4, 5]
         print(min(my_list))
         print(max(my_list))
```

```
1
5
```

3.1.5 Iterating over Lists

In Python, we can directly iterate over a list (in comparison to other programming langues, where we need an specific iterator). Let's print each element in the list.

```
In [23]: my_list = [2, 6, 9, 10]

         for element in my_list:
             print(element)
```

```
2
6
9
10
```

It is therefore not necessary to use `range(len(my_list))` when iterating over lists! This can lead to index errors and the code is much more difficult to understand.

```
In [24]: # don't do this!!
         for i in range(len(my_list)):
             print(my_list[i])
```

```
2
6
9
10
```

Sometimes, we need to iterate over each element in a list, but we do not actually need to use the elements themselves within the loop. In this case, we can use an underscore `_`.

```
In [25]: for i in my_list:  # also working, but not recommended
             print("Hello")

         for _ in my_list:
             print("Hello")
```

```
Hello
Hello
Hello
Hello
```

```
Hello
Hello
Hello
Hello
```

3.1.6 `for` loops with `zip()` and `enumerate()`

We often come across situations where we want to iterate over both the indices and the values of a list. Or we want to iterate over two or more containers at the same time. The naive solution to this problem is to just iterate over the indices and then use square brackets to access the elements. This is not very elegant and is prone to errors. Instead, you should use `enumerate()` and `zip()` to write cleaner code.

```
In [26]: names = ["Alice", "Bob", "Charlie"]
         ages = [22, 19, 23]

         # DON'T DO THIS
         for i in range(len(names)):
             print(f"{names[i]} is {ages[i]} years old")
```

```
Alice is 22 years old
Bob is 19 years old
Charlie is 23 years old
```

Use `zip()` instead.

```
In [27]: for name, age in zip(names, ages):
             print(f"{name} is {age} years old.")
```

```
Alice is 22 years old.
Bob is 19 years old.
Charlie is 23 years old.
```

`zip()` can take any number of arguments.

```
In [28]: semesters = [5, 1, 6]

         for name, age, semester in zip(names, ages, semesters):
             print(f"{name} is {age} years old and studying in semester {semester}.")
```

```
Alice is 22 years old and studying in semester 5.
Bob is 19 years old and studying in semester 1.
Charlie is 23 years old and studying in semester 6.
```

In other situations, you might also need the index of the items while iterating over them. Instead of iterating over just the index and then using the `[]` operator to access the element, you should use `enumerate()`.

```
In [29]: # DON'T DO THIS
         for i in range(len(names)):
             print(f"{names[i]} has index {i}.")
```

```
Alice has index 0.
Bob has index 1.
Charlie has index 2.
```

Use `enumerate()` instead.

```
In [30]: for i, name in enumerate(names):
             print(f"{name} has index {i}.")
```

```
Alice has index 0.
Bob has index 1.
Charlie has index 2.
```

`zip()` and `enumerate()` can also be used together.

```
In [68]: for i, (name, age) in enumerate(zip(names, ages)):
             print(f"{name} is {age} years old and has index {i}.")
```

```
Alice is 22 years old and has index 0.
Bob is 19 years old and has index 1.
Charlie is 23 years old and has index 2.
```

3.1.7 Two-dimensional Lists

As the name linear container implies, lists only have one dimension. However, it is possible to create lists of lists to create a two-dimensional list.

```
In [32]: list_2d = [
             [1, 2, 3],
             [4, 5, 6],
             [7, 8, 9],
             [10, 11, 12],
         ]
         print(list_2d)
```

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

Indexing is done from the outer most list to the inner most list. Let's print the third sublist.

```
In [33]: print(list_2d[2])
```

```
[7, 8, 9]
```

Access the second element of third sublist. Remember: `my_list[index outer list][index inner list]`

```
In [34]: print(list_2d[2][1])  # [third sublist][second element]
```

```
8
```

This concept can also be generalized to higher dimensions.

```
In [35]: list_3d = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]

         print(list_3d)
         print(list_3d[1][0][1])
```

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

3.1.8 Unpacking Assignments

We can assign the elements of list (or any other iterable) to individual variables using unpacking assignments. Let's get the individual words from the list.

```
In [36]: my_words = ["Home", "Tree", "Cat"]
         w1, w2, w3 = my_words
         print(w2)
```

```
Tree
```

Calculate the area of a rectangle. The rectangle is defined by two points in a coordinate system that represent its opposite corners. Use the unpacking assignment.

```
In [37]: coordinates = [[1, 2], [4, 3]]
         point1, point2 = coordinates
         x1, y1 = point1
         x2, y2 = point2
         area = (abs(x2 - x1)) * (abs(y2 - y1))
         print(f"The area ist {area}.")
```

```
The area ist 3.
```

3.2 Tuples

So far, we have used square `[...]` brackets to make collections. However, we can also use regular `(...)` brackets. But what is the difference?

If we use square brackets `[...]`, we create a list, and if we use regular brackets `(...)`, we create a tuple.

While they behave very similarly in a lot of situations, there are some important differences. The most important one is that tuples are immutable. This means that we can't change the individual values in the tuple.

Let's check the type of [1, 2, 3] and (1, 2, 3).

```
In [38]: print(type([1, 2, 3]))
         print(type((1, 2, 3)))
```

```
<class 'list'>
<class 'tuple'>
```

You should use tuples for collections where it does not make sense to change one without changing the other. Use lists for collections that may change individually.

E.g., a city and a postal code belong together. It does not make sense to modify the code.

```
In [39]: city = ("Hannover-Nordstadt", 30167)
         # city[1] = 30168 # not possible
```

It just makes sense to change the entire entry.

```
In [40]: city = ("Hannover-Mitte", 30159)
```

In comparison, for list it makes sense to add or change elements.

```
In [41]: shopping_list = ["milk", "bread", "appels"]
         shopping_list[2] = "bananas"
         shopping_list.append("eggs")
         print(shopping_list)
```

```
['milk', 'bread', 'bananas', 'eggs']
```

Unpacking assignments also work for tuples.

```
In [42]: point = (1, 2)
         x, y = point
         print(x)
```

```
1
```

3.3 Sets

Sets represents an unordered collection of unique elements. Sets are useful when you want to store a group of items but don't care about their order and want to avoid duplicates. You can create a set using curly braces {}.

```
In [43]: my_set = {1, 2, 3, 4, 5}
         print(my_set)
```

```
{1, 2, 3, 4, 5}
```

Since sets can only store unique elements, any duplicates are automatically removed.

```
In [44]: my_set2 = {1, 2, 2}
         print(my_set2)
```

```
{1, 2}
```

Elements in a set cannot be changed. However, they can be removed and new elements can be added.

```
In [45]: my_set = {1, 2, 3}

         my_set.add(4)  # add an element to the set
         print(my_set)

         my_set.remove(2)  # remove an element from the set
         print(my_set)
```

```
{1, 2, 3, 4}
{1, 3, 4}
```

As sets are unordered, we can not use indexing with sets.

3.4 Dictionaries

3.4.1 Working with Dictionaries

Python provides dictionaries, which are a fundamental data structure that associates keys with values. Dictionaries are **unordered** collections of items, and they are defined by key-value pairs: `my_dict = {key: value}`

Let's create a dictionary with different attributes of a person.

```
In [46]: my_dict = {
             "name": "John",
             "age": 30,
             "city": "New York",
         }  # name is the key, 'John' is the value
         print(my_dict)
```

```
{'name': 'John', 'age': 30, 'city': 'New York'}
```

The values can be accessed by using the keys.

Hint: Remember that we need to use single quotation marks ' inside the ***f-string*** *if we use double quotation marks " outside.*

```
In [47]: name = my_dict["name"]
         print(name)
         print(f"Age: {my_dict['age']}")
```

```
John
Age: 30
```

The values of a dictionary can be modified.

```
In [48]: my_dict["age"] = 31
         print(f"Updated Age: {my_dict['age']}")
```

```
Updated Age: 31
```

A new key-value pair can be added.

```
In [49]: my_dict["country"] = "USA"
         print(f"Country: {my_dict['country']}")
         print(my_dict)
```

```
Country: USA
{'name': 'John', 'age': 31, 'city': 'New York', 'country': 'USA'}
```

And a key-value pair can also be removed.

```
In [50]: del my_dict["country"]
         print(my_dict)
```

```
{'name': 'John', 'age': 31, 'city': 'New York'}
```

We can also get a list of all keys, values or items of the dictionary.

```
In [51]: print(list(my_dict.keys()))
         print(list(my_dict.values()))
         print(list(my_dict.items()))
```

```
['name', 'age', 'city']
['John', 31, 'New York']
[('name', 'John'), ('age', 31), ('city', 'New York')]
```

With `.get()`, we can also return an item based on the key. We can set a default value for the case that the key is not present.

```
In [52]: print(my_dict)
         name = my_dict.get("name")
         print(name)
         number = my_dict.get("phone_number", "na")
         print(number)
```

```
{'name': 'John', 'age': 31, 'city': 'New York'}
John
na
```

3.4.2 Loops over Dictionaries

We can loop through all keys in the dictionary and print them.

```
In [53]: for key in my_dict:
             print(key)
```

```
name
age
city
```

With `.values()`, we can loop through all values in the dictionary.

```
In [54]: for key in my_dict.values():
             print(f"{key}")

         # remember: "key" is a variable name, you can choose any name you like
         # however, it is useful to choose a name that makes sense
         # so this would be better
         for v in my_dict.values():
             print(f"{v}")
```

```
John
31
New York
John
31
New York
```

If we want to loop through all keys and values at the same time, we need to use `.items()`.

```
In [55]: for key, value in my_dict.items():
             print(f"{key}: {value}")
```

```
name: John
age: 31
city: New York
```

We can also check if a key is in the dictionary and print its value.

```
In [56]: if "city" in my_dict:
             print(f"The City is {my_dict['city']}.")
         else:
             print("City not in dictionary.")
```

```
The City is New York.
```

3.5 Comparison of Lists, Sets, Tuples and Dictionaries

This table gives you an overview over lists, sets, tuples and dictionaries.

	lists	tuples	sets	dictionaries
ordered?	yes	yes	no	yes (insertion-ordered)
changeable?	yes	no	yes *	yes
indexed?	yes	yes	no	no (keys are used for access)
duplicate elements?	yes	yes	no	no duplicate keys

**elements can be added or removed*

3.6 List and Dictionary Comprehensions

3.6.1 Basic Idea

Comprehensions are a very powerful and elegant language feature in Python. They allow for the generation of a new list or dictionary by applying an expression to each item in an iterable, like a list or a range, in a single line of code. The basic syntax for lists is `mylist = [expression for item in iterable if condition]`, where the `if condition` part is optional. This approach is generally more readable and faster compared to traditional for-loop-based list construction.

3.6.2 List Comprehensions

Let's create a list with the numbers from 0 to 9 as we learned it so far using a `for` loop.

```
In [57]: my_list = []
         for i in range(10):
             my_list.append(i)
         print(my_list)
```

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

This can also be done in one line using a list comprehension.

```
In [58]: my_list_2 = [x for x in range(10)]
         print(my_list_2)
```

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

Another example: create a list of four factories called factory_1, factory_2, ... *Hint: This will become very handy when creating instances...*

```
In [59]: factories = [f"factory_{x}" for x in range(1, 5)]
         print(factories)
```

```
['factory_1', 'factory_2', 'factory_3', 'factory_4']
```

We can use that syntax with the function $f(x) = x$ for x between 0 and 9.

```
In [60]: my_list_3 = [x**2 for x in range(10)]
         print(my_list_3)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can even use a function directly in the list comprehension, for example with the function $f(x) = \frac{2 \cdot x}{5}$. We first define the function:

```
In [61]: def my_func(x):
             return (2 * x) / 5
```

We can then define the list by using a list comprehension with the function.

```
In [62]: my_third_list = [my_func(x) for x in range(10)]
         print(my_third_list)
```

```
[0.0, 0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 3.6]
```

Instead of using range(10), we can also use our previously defined list `my_list`.

```
In [63]: my_fourth_list = [my_func(x) for x in my_list]
         print(my_fourth_list)
```

```
[0.0, 0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 3.6]
```

And we can do the same as above but only use even numbers.

```
In [64]: my_third_list = [my_func(x) for x in my_list if x % 2 == 0]
         print(my_third_list)
```

```
[0.0, 0.8, 1.6, 2.4, 3.2]
```

3.6.3 List Comprehensions for Multidimensional Lists

List comprehensions can also be nested within each other to generate or transform multidimensional lists. Or they can be combined with structured bindings to iterate through multiple lists at once.

In this example, each coordinate corresponds to the product of x and y:

	x = 0	x = 1	x = 2	x = 3	x = 4
y = 0	0	0	0	0	0
y = 1	0	1	2	3	4
y = 2	0	2	4	6	8
y = 3	0	3	6	9	12

	x = 0	x = 1	x = 2	x = 3	x = 4
y = 4	0	4	8	12	16

Let's create a two-dimensional list based on the table above (one list for each row) using a list comprehension.

```
In [65]: list_2d = [[x * y for x in range(5)] for y in range(5)]

         print(list_2d)
         print(list_2d[2])
```

```
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8], [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]
[0, 2, 4, 6, 8]
```

3.6.4 Dictionary Comprehension

The equivalent of list comprehensions also exists for dictionaries. Let's generate a dictionary, in which every name has his corresponding number of characters.

```
In [66]: my_keys = ["Tim", "Tom", "Tina"]

         my_dict = {key: len(key) for key in my_keys}
         print(my_dict)
```

```
{'Tim': 3, 'Tom': 3, 'Tina': 4}
```

Another example: Create a dictionary of production quantities for factories. The dictionary should be named `factory_x` for x an include the factories from 1 to 5. The production quantity is defined as x^2 for each x.

```
In [67]: production_quantities = {f"factory_{x}": x**2 for x in range(1, 6)}

         print(production_quantities)
```

```
{'factory_1': 1, 'factory_2': 4, 'factory_3': 9, 'factory_4': 16, 'factory_5': 25}
```

Chapter 4

Modules

4.1 Loading Modules

Modules are files that contain Python code (like a code library). They make it possible to split code into smaller parts, which improves maintainability and reusability. Later on, we will learn how to create our own modules. For now, we will learn how to work with existing modules.

Modules are loaded with `import` within our code. It is good practice to place **all import statements at the top** of a file.

Let's import the module `datetime`.

```
In [1]: import datetime
```

If a module is not used in the following code, VS Code displays the module name in light gray. This changes when we use the module. Let's print out the current date and time to see this effect. You should only load modules that you are really using to keep your code clean.

```
In [2]: print(datetime.datetime.now())  # current time
```

```
2024-10-25 08:41:57.704280
```

If the module name is long, we can use an alias with `as` (`import module_name as alias`). Please note that there are common abbreviations for some modules that should be used for readability (e.g. for numpy → np or pandas → pd).

```
In [3]: import datetime as dt

        print(dt.datetime.now())  # current time
```

```
2024-10-25 08:42:10.835358
```

4.2 Standard Modules in Python

The standard library in Python contains a number of useful modules that do not need to be installed separately.

You can find an overview at : https://docs.python.org/3/library/index.html

You do not need to memorize all the individual packages. Programming often involves searching online for solutions when you need to implement a specific feature. Sometimes, a suitable module is already available for use.

Let's have a look at some frequently used packages.

1. `math`: For mathematical functions

```
In [4]: my_list = [1, 2, 3, 4, 5]
        # the import statement is just in this cell for teaching purpose, place it in␣
↪one cell at the top!
        import math

        print(math.prod(my_list))  # product of all elements
        print(math.pow(2, 4))  # 2^4
        print(math.factorial(6))  # 5!
        print(math.pi)  # pi
```

```
120
16.0
720
3.141592653589793
```

2. `random`: Random numbers (***Hint***: *Look at this package for yourself in more detail. It is very useful for instance generation. Check out, why you should always use a seed!*)

```
In [5]: import random

        print(random.random())  # random number between 0 and 1
        print(random.choice(my_list))  # random choice from list
```

```
0.7333950229608168
1
```

3. `timeit`: Measure the time of functions

```
In [1]: def example_function():
            total = 0
            for i in range(100000):
                total += i
            return total


        import timeit

        # globals() is used to pass the global variables and functions to timeit
        # number is the number of times the function is called
        execution_time = timeit.timeit("example_function()", globals=globals(),␣
↪number=1000)

        print(f"The execution time is: {round(execution_time, 2)} seconds")
```

```
The execution time is: 7.77 seconds
```

4. ... *There are many more*

We will learn more about the most common modules like `os`, `pickle` and `sys` later on.

4.3 External Modules: Installing and Using Modules Outside the Standard Library

External modules (or packages) are not part of the Python standard library. Thus, they need to be installed separately. This can be done via the Python package manager `pip`.

Open a terminal on your computer. Than, simply type `pip install package_name`.

E.g. for using the package `numpy`, type `pip install numpy`.

To get a list with all installed packages, type `pip list`.

During this course, we will work especially with - `otter-grader` - `numpy` - `pandas` - `matplotlib` - `gurobipy` - `networkX`

Note: Sometimes problem occur because there are several Python versions on a computer and the wrong version is then used with pip. You find your current Python interpreter at the top right corner of VS Code (should be Python 3.11/3.12/3.13).Try ***`python-[version] -m pip install [package_name]`*** *to install a package in a specific python version.*

4.4 Loading Parts from Modules

Modules are often very extensive. It can therefore sometimes make sense to only load certain functions from a package. We can do this with `from module import function`.

Let's look an an example with the module `itertools`. We import `combinations` to get all pairwise combinations in the list.

```
In [6]: data = ["apple", "banana", "cherry", "strawberry"]

        from itertools import combinations

        combs = combinations(data, 2)
        for comb in combs:
            print(comb)
```

```
('apple', 'banana')
('apple', 'cherry')
('apple', 'strawberry')
('banana', 'cherry')
('banana', 'strawberry')
('cherry', 'strawberry')
```

If we now want to use another class from `itertools`, we need to import it as well. Otherwise, the code does not work. We can also use `as` in this context. (*Note: In this example, the* `c` *is just used for teaching purposes. Make sure you still use specific abbreviations.*)

```
In [ ]: from itertools import count as c  # only works with this line
```

```
        for number in c(start=1, step=2):
            if number > 10:
                break
            print(number)
```

Consequently, for this example, we could use one of the following statement:

```
In [ ]: from itertools import combinations
        from itertools import count as c

        combs = combinations(data, 2)
        for comb in combs:
            print(comb)


        for number in c(start=1, step=2):
            if number > 10:
                break
            print(number)
```

```
In [ ]: from itertools import combinations, count

        combs = combinations(data, 2)
        for comb in combs:
            print(comb)


        for number in count(start=1, step=2):
            if number > 10:
                break
            print(number)
```

```
In [ ]: import itertools

        combs = itertools.combinations(data, 2)
        for comb in combs:
            print(comb)


        for number in itertools.count(start=1, step=2):
            if number > 10:
                break
            print(number)
```

Or we can even import the entire module and in addition explicitly import a specific function/class. This reduces the writing work somewhat if we need this one thing very often. In this example, we do not have to write `itertools.count`, but only `c`. All other elements of the `itertools` module can still be used via `itertools.name`.

```
In [ ]: import itertools
        from itertools import count as c
```

```
        combs = itertools.combinations(data, 2)
        for comb in combs:
            print(comb)

        for number in c(start=1, step=2):
            if number > 10:
                break
            print(number)
```

It is also possible to import everything with `*`. However, this is **not** recommended as you do not know, where `count` and `combinations` come from.

```
In [ ]: from itertools import *

        combs = combinations(data, 2)
        for comb in combs:
            print(comb)


        for number in count(start=1, step=2):
            if number > 10:
                break
            print(number)
```

4.5 Example

Imagine that this is an example of a new jupyter notebook or plain python file (more about that later). Place all loaded modules at the top and than your code afterwards.

```
In [14]: import math
         import random
         import os  # Don't include the module if you are not using it
```

```
In [15]: def calculate_random_circle_area(radius):
             return math.pi * radius**2
```

```
In [ ]: # do not place "import random" here
        radius = random.randint(1, 10)
        area = calculate_random_circle_area(radius)

        print(f"The area for r = {radius} is {area}.")
```

Chapter 5

Paths and Files

```
In [3]: import os
```

5.1 Absolute and Relative Paths

A path specifies the location of directories or files. It usually consists of

- a drive or disk label (e.g. in Windows `C`)
- a list of directories (folder and subfolder)
- a file name (e.g. calculator.py)

An example for a path could be: `C:/Users/documents/homework/calculator.ipynb`
We distinguish between absolute and relative paths.
Source and more information: https://lehre.idh.uni-koeln.de/lehrveranstaltungen/wisem21/it-zertifikat-der-phil.fak-advanced-web-basics-7/web-technologien-1/html-1/relative-vs-absolute-pfade/
An **absolute path** defines the complete position of a file on a specific computer. The starting point is the root directory (e.g. `C` in Windows). An example of an absolute path is:

```
In [4]: path_abs = "C:/Users/documents/homework/calculator.py"
```

The problem with absolute paths is that they usually only work on the computer on which the program was developed, as the path is specific to this directory structure.
An alternative is the use of **relative paths**. Relative paths have the current working directory as their origin.
Let's look up the current working directory using `os` for this Jupyter Notebook.

```
In [5]: print(f"The current working directory is {os.getcwd()}")
```

```
The current working directory is /workspaces/imku2023/src
```

If we we need to go one level up in a relative path, we use `..` An example of a relative path is:

```
In [6]: path_rel = "../results/myfile.txt"
```

In most cases, you should use relative paths!

5.2 Directories

To organize files properly, it is useful to work with directories. Let's create a directory called `this_is_a_test` using `os.mkdir()`.

```
In [7]: os.mkdir("this_is_a_test")
```

Let's check what happens if we run the code again.

```
In [8]: # os.mkdir('this_is_a_test') # uncomment to see the error
```

As the directory already exists, we are running into an error. So let's delete the directory.

```
In [9]: import shutil

        shutil.rmtree("this_is_a_test", ignore_errors=True)
```

However, we do not always want to delete an entire directory, as there could be subdirectories, for example, that should be retained. Therefore, when creating directories, you should **always** check whether the directory already exists using `os.path.exists()` and than only create the directory if it not exists.

```
In [10]: if not os.path.exists("this_is_a_test"):
             os.mkdir("this_is_a_test")

         # even better would be:
         # path = "this_is_a_test"
         # if not os.path.exists(path):
         #    os.mkdir(path)
```

This code can now be executed multiple times without an error.

We often have subdirectories in a folder. Imagine you are solving a model with several instances. You want to save the results for each instance in a separate directory. In this case, a top directory and subdirectories must be created.

We use `os.makedirs()` to create multiple directories. Check again if the path already exists to avoid errors.

```
In [11]: directory = "this_is_a_test_2"
         instances = ["case_1", "case_2"]

         for instance in instances:
             # define the variable path, as it will be used multiple times
             path = f"{directory}/{instance}"
             if not os.path.exists(path):
                 os.makedirs(path)
```

During the rest of the course, we will work with the following directory structure: - lecture - exercise - data - results

In data, we will save all our generated data as it will be (sometimes) used in multiple lectures.

Create the two directories `data` and `results`. For `data`, create a subdirectory called `lecture_2`.

```
In [12]: data_path = "../data/lecture2"
         if not os.path.exists(data_path):
             os.makedirs(data_path)
```

```
result_path = "../results"
if not os.path.exists(result_path):
    os.makedirs(result_path)
```

5.3 Text Files

5.3.1 Accessing Text Files

A simple way for storing data is a text file (`.txt`). Let's create a text file called `MyNumbers.txt` with the numbers from 1 to 8 in the directory `lecture2`. We use `open()` to open the file with the option `w` for writing. `\n` will create a line brake and at the end, the file is closed with `.close()`.

```
In [33]: file = open(f"{data_path}/MyNumbers.txt", mode="w")
         for i in range(1, 9):
             file.write(f"{i}\n")  # convert to a string
         file.close()
```

File Access Modes specify how the file will be used. The most common modes are: - read only (`r`) - write only (`w`) - append only (`a`)

We can also load the file and print the content using `r`.

```
In [34]: file = open(f"{data_path}/MyNumbers.txt", mode="r")
         print(file.read())
         file.close()
```

```
1
2
3
4
5
6
7
8

```

When using (German) words, you might encounter problems with encoding, e.g. Spaßvogel might be displayed as SpaÃŸvogel. You can define the encoding option in this case with `f = open(my_path, mode='r', encoding='utf-8')`.

An alternative to `open()` and `close()` is `with open()`. It will automatically closes the file for you. However, the entire code has to be indented by one tab.

```
In [35]: with open(f"{data_path}/MyNumbers.txt", mode="r") as f:
             print(f.read())
```

```
1
2
3
4
5
6
```

```
7
8
```

Why should a file be closed at the end? Often, your code is also working when you have not closed a file. Still there are many good reasons to do so, see https://realpython.com/why-close-file-python/#in-short-files-are-resources-limited-by-the-operating-system.

5.3.2 Reading Files Line by Line

In most cases, you want to save your loaded information from the text file in a variable to do something with it. There are multiple ways for doing so: - using a `for` loop - `read()` and `splitlines()` - ... (you can google more)

Create the file `companies.txt` with the names of four companies with each line containing one name.

```
In [16]: company_data_path = f"{data_path}/companies.txt"
         file = open(company_data_path, mode="w")
         for i in range(1, 5):
             file.write(f"Company_{i}\n")
         file.close()
```

1. We can now read the file. Let's print each line using a `for`-loop and add the companies to a list.

```
In [17]: companies = []
         with open(company_data_path, mode="r") as f:
             for line in f:
                 print(line)
                 companies.append(line)
         print(companies)
```

```
Company_1

Company_2

Company_3

Company_4

['Company_1\n', 'Company_2\n', 'Company_3\n', 'Company_4\n']
```

To leave out the line separator `\n`, we can use `rstrip()`.

```
In [18]: companies = []
         with open(company_data_path, mode="r") as f:
             for line in f:
                 print(line.rstrip())
                 companies.append(line.rstrip())
         print(companies)
```

```
Company_1
Company_2
```

```
Company_3
Company_4
['Company_1', 'Company_2', 'Company_3', 'Company_4']
```

It is even better to use a list comprehension to save our companies in the list.

```
In [19]: with open(company_data_path, mode="r") as f:
             my_companies = [line.rstrip() for line in f]
         print(my_companies)
```

```
['Company_1', 'Company_2', 'Company_3', 'Company_4']
```

Of course, instead of `with open()`, it works also with `open()` and `.close()`.

```
In [20]: file = open(company_data_path, mode="r")
         my_companies = [line.rstrip() for line in file]
         file.close()
         print(my_companies)
```

```
['Company_1', 'Company_2', 'Company_3', 'Company_4']
```

2. Another option to read files line by line is the use of `read()` and `splitlines()`. `splitlines()` will split the string into a list. The splitting is done at line breaks (`\n`).

```
In [21]: with open(company_data_path, mode="r") as f:
             my_companies = f.read().splitlines()
         print(my_companies)
```

```
['Company_1', 'Company_2', 'Company_3', 'Company_4']
```

5.4 CSV-Files

Data can also be stored in `CSV`-files. `CSV` stands for Comma-separated values. You can read and write `CSV`-files with the `csv`-module. `CSV`-files can also be opened into Excel (just google it, if your are interested).

Import the `csv`-module:

```
In [22]: import csv
```

Assume we have some factory data given.

```
In [23]: import random

         random.seed(1)

         production_quantities = {f"company_{i}": random.randint(1, 10) for i in␣
↪range(1, 5)}
```

Let's write our factory data to a CSV-files. We write a new row with `.writerow()`. The standard separator is a comma. If you need another delimiter, you can define it within `csv.writer()`.

```
In [24]: path = "../data/lecture2/factory_data.csv"

         with open(path, mode="w", newline="") as csv_file:
             csv_writer = csv.writer(csv_file)
             csv_writer.writerow(["Factory", "Quantity"])
             for factory, quantity in production_quantities.items():
                 csv_writer.writerow([factory, quantity])
```

We can also load the data from the CSV-file.

```
In [25]: with open(path, mode="r") as csv_file:

             f = csv.reader(csv_file)

             for factory, quantity in f:  # unpacking assignment
                 print(factory)
                 print(quantity)
```

```
Factory
Quantity
company_1
3
company_2
10
company_3
2
company_4
5
```

The CSV-files can be useful for saving results, especially if you want to quickly view your results with Excel or need to create an Excel file. However, for data, it is often better to preserve the data structures and use `pickle`.

5.5 Pickle Files

In many cases, we want to save generated data so that it can be used again at another place. For example, let's assume that we have generated a list of factories and production quantities. We could now write them into text files and load them again, for example in another file, when creating a model. However, we would lose the already existing data structure.

An alternative is `pickle`. With `pickle`, we can just use `dump()` to save our data and `load()` to load it in another file. In comparison to text files, we cannot open a pickle file in an editor. The advantage is that we can use `pickle` with any data type we like (lists, dicts, custom classes, etc.).

Let's create a list with five factories and a dictionary with random production quantities for each factory.

```
In [26]: import random
         import pickle

         factories = [f"factory_{x}" for x in range(1, 4)]
         production_quantities = {factory: random.randint(1, 9) for factory in factories}
```

We now use `pickle` to save our data. There are two options: We can create two separate pickle files, one for the factories and one for the production quantities. Alternatively, we can create a dictionary that includes both factories and production quantities. In this case, we will just have one file. Let's use the second option.

Create one dictionary `data` containing all the data:

```
In [27]: data = {"factories": factories, "quantities": production_quantities}
         print(data)
```

```
{'factories': ['factory_1', 'factory_2', 'factory_3'], 'quantities': {'factory_1': 2,␣
 ↪'factory_2': 8, 'factory_3': 8}}
```

Now use `pickle.dump()` to save the data. For `pickle`, you can just choose any file extension you like. Here, the file extension would be `factory`.

```
In [28]: pickle_path = "../data/lecture2/factory_data.factory"

         with open(pickle_path, "wb") as file:  # wb for write binary
             pickle.dump(data, file)
```

Now, we can load the data again with `pickle.load()`. This would also work in a completely different Jupyter/Python project.

```
In [29]: with open(pickle_path, "rb") as f:
             loaded_data = pickle.load(f)
         print(loaded_data)
         print(loaded_data["factories"])
         print(loaded_data["quantities"])
```

```
{'factories': ['factory_1', 'factory_2', 'factory_3'], 'quantities': {'factory_1': 2,␣
 ↪'factory_2': 8, 'factory_3': 8}}
['factory_1', 'factory_2', 'factory_3']
{'factory_1': 2, 'factory_2': 8, 'factory_3': 8}
```

We recommend using `pickle` when working with Python, especially at the beginning. It is easy to use and preserves your data structure.

However, keep in mind that `pickle` is Python-specific. So if you work with different languages, such as C++, you will not be able to use your files. Also, `pickle` can have safety risks when you are working with files from unknown sources. There are also other formats for saving data, like `JSON`, that you can look at in those cases.

For this course and for most of your use cases, however, `pickle` is a useful way to save data.

5.6 Notes on Paths and Path Separators

Path separators: Windows systems use a backslash \ as path separator while Linux systems use a slash /. If you just copy a Windows path out of a directory, it leads to an error. Why? The \ is the start of an escape sequence, presenting not the character itself but another character or series of characters.

```
In [30]: # Path leading to an error
         # path_dir = 'C:\Users\documents\homework\calculator.py'
```

```
        # escape sequences
        print("Next \n comes a line break and a \new line.")
        # how to print a '\'?
        print("We print a backslash as \\, e.g. \\new.")
```

```
Next
 comes a line break and a
ew line.
We print a backslash as \, e.g. \new.
```

There are multiple ways to make sure that paths are working on all operating systems. The easiest is to use the Linux separator `/`, it will also work on Windows.

```
In [31]: path = "C:/Users/documents/homework/calculator.py"

        # instead of
        # path_windows_original = 'C:\Users\documents\homework\calculator.py'
```

Or you can also use a raw string with ‘`r`, so that special characters will not be evaluated.

```
In [32]: # path_raw = r"C:\Users\documents\homework\calculator.py"
```

In addition, there are also packages or a function from `os` to deal with these problems.

It is likely that you will encounter issues related to paths while programming. For instance, certain editors may set the current working directory differently from others. Additionally, our own experience has indicated that paths on the LUH cluster are (sometimes) managed differently compared to those on our local computers. Problems can also arise when working with modules or when executing code on different operating systems. In all these cases, there is not always a very clear way of finding a solution. Nevertheless, it helps to check from which location the programs are executed.

Chapter 6

Functions

6.1 Basics

We have already learned the first basics of functions:

```
In [2]: def maximum(a, b):
            if a > b:
                return a
            else:
                return b


        print(maximum(2, 3))
        print(maximum(6, 5))
```

```
3
6
```

A function in Python is a block of reusable code that performs a specific task. Functions help to structure and modularize the code. You can define a function once and then use it everywhere in your program instead of writing the same code multiple times. A function starts with `def name():`

Remember: The names of the parameters in a function **do not need to match the names** of the variables passed to it, as they are specific for the function. A function can **return** none, one or multiple value(s). You can save the returned values in variables using **unpacking assignments**.

```
In [3]: def calculate(a, b):
            return a + b, a * b
```

```
In [4]: a = 2
        b = 3
        print(calculate(a, b))

        first_number = 1
        second_number = 3
        sum, mul = calculate(first_number, second_number)
        print(sum)
        print(mul)
```

```
(5, 6)
4
3
```

6.2 Global and Local Variables

Variables have a specific scope of validity: *local variables* are created within a function and can only be used in this function. *Global variables* that are created outside a function and are accessible throughout the entire program. Let's look at an example for a local and a global variable.

```
In [5]: number = 5


        def my_func():
            print(number * number)

            word = "Python"
            print(word)


        my_func()
        # print(word) # will raise an error, because the variable word is not defined␣
↪outside the function
```

```
25
Python
```

The example above shows that we can access the global variable within in the function.

However, we cannot change the global variable within the function, see the example below: With `number=3`, we create a new local variable `number` in the function. Thus, `number` will be considered as local variable in the entire function. Since the assignment comes after the `print()` call, the local variable `number` is not defined at this point, which leads to an error.

```
In [6]: number = 5


        def my_func():
            print(number * number)
            # number = 3 # leads to an error


        my_func()
```

```
25
```

If we just want a local variable and not consider the global variable, we can change the order. The assignment of `number = 3` has no effect on the global variable in this case.

```
In [7]: number = 5


        def my_func():
            number = 3
            print(number * number)


        my_func()
        print(number)
```

```
9
5
```

If we want to change the global variable in the function, we can use the keyword `global`.

```
In [8]: number = 5


        def my_func():
            global number
            print(number * number)
            number = 3


        my_func()
        print(number)
```

```
25
3
```

Overall, global variables should be avoided wherever possible. They make debugging more difficult and the code less clear. It is better to pass parameters to functions and work with returns!

```
In [9]: # Better avoid using global variables
        # Use function arguments and return values instead

        number = 5


        def my_func(n):
            print(n * n)
            n = 3
            return n


        number = my_func(number)
        print(number)
```

```
25
3
```

6.3 Pass by Value vs. Pass by Reference

There are basically two concepts for passing arguments to functions: *"by value"* and *"by reference"*. 1. *"by value"*: For immutable objects such as `int`, `float`, `str` and `tuple`, **a copy of the value** is passed to the function. Changes within the function do not affect the original object. 2. *"by reference"*: For mutable objects such as `list` and `dict`, **a reference** to the original object is passed to the function. Changes within the function affect the original object.

Let's look at an example for an `int` and a `string`. In this example, the number and the string do not change, as they are passed by value.

```
In [10]: my_number = 5
         word = "hello"


         def do_something(my_number, word):
             my_number = 1
             word = "Goodbye"


         do_something(my_number, word)

         print(my_number)
         print(word)
```

```
5
hello
```

If an element is added to a list in a function, the original list is changed as the mutable list is passed by reference.

```
In [11]: my_list = [1, 2, 3]


         def change_something(my_list):
             my_list.append(4)


         change_something(my_list)
         print(my_list)
```

```
[1, 2, 3, 4]
```

However, if we define a new list within the function, the new list does not affect the original list (even though they have the same name).

```
In [12]: my_list_2 = [7, 8, 9]


         def change_something_2(my_list_2):
             my_list_2 = my_list_2 + [10, 11]


         change_something_2(my_list_2)

         print(my_list_2)
```

```
[7, 8, 9]
```

If we want to work with the new list, we need to return it. The same applies to immutable objects, such as integers.

```
In [14]: my_list_2 = [7, 8, 9]
         my_number = 5


         def change_something_2(my_list_2):
             my_list_2 = my_list_2 + [10, 11]
             my_number = 1
             return my_list_2, my_number


         print(my_number)
         print(change_something_2(my_list_2))
```

```
5
([7, 8, 9, 10, 11], 1)
```

Sometimes, we do not want the original list to be changed within the function and keep the original list unchanged. We can than just manually pass a copy with `.copy()` of the list in the function.

```
In [ ]: my_list_3 = [1, 2, 3]


        def change_my_list_3(my_list_3):
            my_list_3.append(4)
            print(f"Within function: {my_list_3}")


        change_my_list_3(my_list_3.copy())

        print(f"Outside function: {my_list_3}")
```

```
Within function: [1, 2, 3, 4]
Outside function: [1, 2, 3]
```

However, we can encounter a problems with nested structures, such as dictionaries: if we use `.copy()` on the dictionary, only the top level dictionary will be copied (shallow copy). The included dictionary is passed as an reference.

```
In [ ]: def nested_change(my_dict):
            my_dict[4] = 300
            my_dict[1][2] = 100
            print(f"Within function: {my_dict}")


        my_dict = {1: {2: 2, 3: 1}, 4: 1}
        nested_change(my_dict.copy())
        print(f"Outside function: {my_dict}")
```

```
Within function: {1: {2: 100, 3: 1}, 4: 300}
Outside function: {1: {2: 100, 3: 1}, 4: 1}
```

We can solve this problem by using `deepcopy` to create a copy of all levels. We have to import the module `copy`. (https://docs.python.org/3/library/copy.html).

```
In [ ]: my_dict = {1: {2: 2, 3: 1}, 4: 1}

        import copy

        nested_change(copy.deepcopy(my_dict))
        print(f"Outside function with deepcopy: {my_dict}")
```

```
Within function: {1: {2: 100, 3: 1}, 4: 1}
Outside function with deepcopy: {1: {2: 2, 3: 1}, 4: 1}
```

Of course, `deepcopy` is not only specific to functions and dictionaries. We can also use it to create a deep copy of a nested list within our code.

```
In [ ]: original_list = [[1, 1, 1], [1, 1]]

        # shallow copy of the list
        shallow_copy = copy.copy(original_list)
        shallow_copy[0][0] = 2
        print(original_list)

        original_list = [[1, 1, 1], [1, 1]]
        # deep copy of the list
        deep_copy = copy.deepcopy(original_list)
        deep_copy[1][0] = 2
        print(original_list)
```

```
[[2, 1, 1], [1, 1]]
[[2, 1, 1], [1, 1]]
```

6.4 Function Parameters

Parameters can be passed as *positional arguments* and as *keyword arguments*. Positional arguments are passed to a function in the order of the parameters defined. Keyword arguments specify the parameter name explicitly, allowing arguments to be passed in **any order**. Combining both is possible, but positional arguments must precede keyword arguments.

```
In [ ]: def add(a, b, c):
            return a + b + c


        n1 = 3
        n2 = 2
        n3 = 1

        print(add(n1, n2, n3))
        print(add(a=n1, c=n3, b=n2))
        print(add(n1, n2, c=n3))
        # print(add(n1, b=n2, n3)) # not working positional argument follows keyword␣
 ↪argument
```

```
6
6
6
```

Functions can have default values for parameters. These parameters can be omitted when calling the function. However, they can also be replaced with other values.

```
In [ ]: def greet(name, greeting="Good morning"):
            print(f"{greeting}, {name}!")


        greet("Bob")  # with default greeting

        greet("Alice", "Guten Morgen")  # with custom greeting
```

```
Good morning, Bob!
Guten Morgen, Alice!
```

If we have several default values, we must specify which value we want to set with a user-defined parameter.

```
In [ ]: def f2(a=1, b=2, c=3):
            print(f"f2: a={a} b={b} c={c}")


        f2()
        f2(4)  # pass the first of the defaults
        f2(b=9)  # only specify a value for b
```

```
f2: a=1 b=2 c=3
f2: a=4 b=2 c=3
f2: a=1 b=9 c=3
```

So far, we have passed the parameters to the function separately. However, we can also summarize parameters as a `tuple` and pass them. To unpack the variables automatically, we use the unpack operator `*`.

```
In [ ]: def f1(a, b, c):
            print(f"f1: {a = } {b = } {c = }")


        f1(1, 2, 3)  # the 'normal' way

        params = (4, 5, 6)
        f1(*params)  # unpack the tuple
```

```
f1: a = 1 b = 2 c = 3
f1: a = 4 b = 5 c = 6
```

Similarly, a dictionary can also be passed and unpacked automatically with `**`.

```
In [ ]: kwargs = {"c": 7, "b": 8, "a": 9}  # the order in the dict does no matter
        f1(**kwargs)  # unpack the dict
```

```
f1: a = 9 b = 8 c = 7
```

We can do the same in reverse and design functions that accept a *undefined number* of arguments in a generated tuple.

```
In [ ]: def f3(*args):
            print(f"f3: args={args}")


        f3()
        f3(1, 2, 3)
```

```
f3: args=()
f3: args=(1, 2, 3)
```

This function accepts one fixed position parameter `a` as well as any number of additional position and keyword arguments. The first position parameter passed will always be assigned to parameter `a`. All additional position arguments will be assigned to the tuple `*args` and all keyword arguments will be assigned to `**kwargs`.

```
In [ ]: def f4(a, *args, **kwargs):
            print(f"f4: a={a} args={args} kwargs={kwargs}")
```

```
f4(1)
f4(1, 2)
f4(1, 2, 3, d=4, e=5)
```

```
f4: a=1 args=() kwargs={}
f4: a=1 args=(2,) kwargs={}
f4: a=1 args=(2, 3) kwargs={'d': 4, 'e': 5}
```

`*args` and `**kwargs` are useful, as we sometimes don't know in advance how many arguments will be passed to the function. Here is an example of a function that adds any number of numbers.

```
In [ ]: def my_sum(*args):
            sum = 0

            for n in args:
                sum = sum + n

            print(f"The sum is: {sum}")


        my_sum(1, 2)
        my_sum(1, 2, 3)
        my_sum(1, 2, 3, 4, 5)
```

```
The sum is: 3
The sum is: 6
The sum is: 15
```

6.5 Early Returns

You will often have a pattern where you need to check that multiple things are okay before doing the actual work. A bad way to do this is to keep nesting `if` statements.

```
In [ ]: def make_even(x):
            if x is not None:
                if x % 2 != 0:
                    # image this line is a very complicated block of code
                    return "make this number even"
                else:
                    return "even"

            return None


        print(make_even(x=3))
        print(make_even(x=None))
        print(make_even(x=4))
```

```
not even
None
even
```

As you can see, this will quickly lead to a lot of indentation and make the code hard to read. Instead, you should use early returns. This means that you check for error cases first and return early if you find one. This will make the code much easier to read.

```
In [ ]: def better_make_even(x):
            if x is None:
                return None

            if x % 2 == 0:
                return "even"

            # image this line is a very complicated block of code
            return "make this number even"


        print(better_make_even(x=3))
        print(better_make_even(x=None))
        print(better_make_even(x=4))
```

```
not even
None
even
```

6.6 Match-Case

We often have different cases and depending on the case, something different should be done, as in the following example of the traffic light. We can do this with `if- else`:

```
In [ ]: def traffic_light_action(color):
            if color.lower() == "red":
                print("Stop")
            elif color.lower() == "yellow":
                print("Get ready")
            elif color.lower() == "green":
                print("Go")
            else:
                print("Invalid color")
```

An alternative is the `Match-Case` statement, which was introduced in Python 3.10.

```
In [ ]: def traffic_light_action(color):
            match color.lower():
                case "red":
                    print("Stop")
                case "yellow":
```

```
                print("Get ready")
            case "green":
                print("Go")
            case _:
                print("Invalid color")


    traffic_light_action("RED")
```

```
Stop
```

`Match-Case` makes the code a little easier to read. However, `if- else` can be used just as well.

We have included this example for two main reasons: 1. There is not always one way that is necessarily the better choice. 2. New versions of Python provide new possibilities. It is therefore a good idea to keep your system up to date.

6.7 Lambda Function

A lambda function is a small, anonymous function that is created with the `lambda` keyword. It is made for cases where using the `def` syntax would create unpleasant code.

The syntax is:
`symbol = lambda parameter_list: return_value`
This is equivalent to: `def symbol (parameter_list):  return return_value`

```
In [ ]: i = 1
        j = 2


        def add_function(x, y):
            return x + y


        result = add_function(i, j)
        print(f"Result function: {result}")

        add = lambda x, y: x + y
        result = add(i, j)
        print(f"Result lambda function: {result}")
```

```
Result function: 3
Result lambda function: 3
```

Why do we need a lambda function? We can use lambdas to pass a function in another function. Let's create a function that can apply any operation to the elements of a list. We want to calculate the sum, product and maximum of the elements in the list `numbers`.

```
In [ ]: numbers = [1, 4, 2, 3]


        def calculate(numbers, func, initial_value):
```

```
            result = initial_value
            for n in numbers:
                result = func(n, result)
            return result


        print("sum", calculate(numbers, lambda x, y: x + y, 0))
        print("prod", calculate(numbers, lambda x, y: x * y, 1))
        print("max", calculate(numbers, max, 0))  # uses the build-in max function
```

```
sum 10
prod 24
max 4
```

A common use case for using lambdas is the build-in sort method `sorted`, but with a self-defined pattern. The syntax is `sorted(iterable, key=*key*, reverse=*reverse*)`, whereas `key`and `reverse` are optional.

Remember: The default sorting is done alphabetically:

```
In [ ]: word_list = ["Elephant", "Cat", "Dragon", "House", "Book"]

        print(sorted(word_list))
```

```
['Book', 'Cat', 'Dragon', 'Elephant', 'House']
```

Let's create a custom function for sorting the words according to their length. This can be done with a normal function.

```
In [ ]: def get_length(word):
            return len(word)


        sorted_word_list = sorted(word_list, key=get_length)
        print(sorted_word_list)
```

```
['Cat', 'Book', 'House', 'Dragon', 'Elephant']
```

However, it is less code if this is done with a `lambda` function.

```
In [ ]: sorted_word_list = sorted(word_list, key=lambda x: len(x))
        print(sorted_word_list)
```

Be careful: passing functions to other functions is part of the functional programming paradigm - it can lead to very elegant code, but in some situations it is much more difficult to understand!

6.8 Documenting Functions

Docstrings, short for documentation strings, are suitable for the documentation of functions. It is about documenting what a function does (and not how). A docstring is started with triple quotes (either double `"""` or single `'''`).

What should a docstring look like?

- The doc string line should begin with a capital letter and end with a period.
- The first line should be a short description.
- If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
- The following lines should be one or more paragraphs describing the object's calling conventions, side effects, etc.

Source: https://www.geeksforgeeks.org/python-docstrings/

When hovering over a function, VS Code displays the docstring in the tooltip, which offers quick reference options.

Let's document a simple function that does nothing.

```
In [ ]: def my_function():
            """This function does nothing"""
            pass
```

In most cases, we have functions that take parameters and return values:

```
In [ ]: def greet(name, greeting="Good Morning"):
            """
            Generates a welcome message.

            Parameter:
                name (str): Name of the person to be greeted.
                greeting (str, optional): The greeting to be used. Defaults to 'Hallo'.

            return value:
                str: a combined welcome message with the greeting and the name.
            """
            return f"{greeting}, {name}!"


        print(greet("Anna"))
        print(greet("Anna", "Hallo"))
```

```
Good Morning, Anna!
Hallo, Anna!
```

In addition to docstrings, you can use type hinting. It indicates the type of a value in Python. The types are than also shown when hovering over the function.

```
In [ ]: def greet_2(name: str, greeting="Good Morning") -> str:
            return f"{greeting}, {name}!"


        def calculate(a: int, b: int) -> int:
            return a + b
```

Together, docstrings and type hinting look like this:

```
In [ ]: def greet(name: str, greeting="Good Morning") -> str:
            """
```

```
    Generates a welcome message.

    Parameter:
        name (str): Name of the person to be greeted.
        greeting (str, optional): The greeting to be used. Defaults to 'Hallo'.

    return value:
        str: a combined welcome message with the greeting and the name.
    """
    return f"{greeting}, {name}!"
```

As it is part of good programming, we require you to document all functions in your exams!
Please understand that we do not document smaller functions in the lecture notebooks in order to focus on the essentials and save time when writing.

Chapter 7

Git

7.1 Installation and Setup

Before we can work with Git, you have to install it on your PC. We recommend this website to download Git: https://git-scm.com/download/win .

After downloading and installing Git you need to access your GitLab-Account. This is done by accessing GitLab from the LUH via this website https://gitlab.uni-hannover.de/users/sign_in . You can login with your normal WebSSO-Account. If you are doing this for the first time you should get asked to set a password for your account. This is not and should not be the same password as for your WebSSO-Account. This password is used for the work with repositories.

Now we want to log in on our PC. We open the terminal by using the windows search bar with the keyword “cmd”. After opening write everything without "" in the cmd-terminal: 1. git config –global credential.helper store 2. git config –global user.name “Your Name” 3. git config –global user.email “youremail@yourdomain.com”

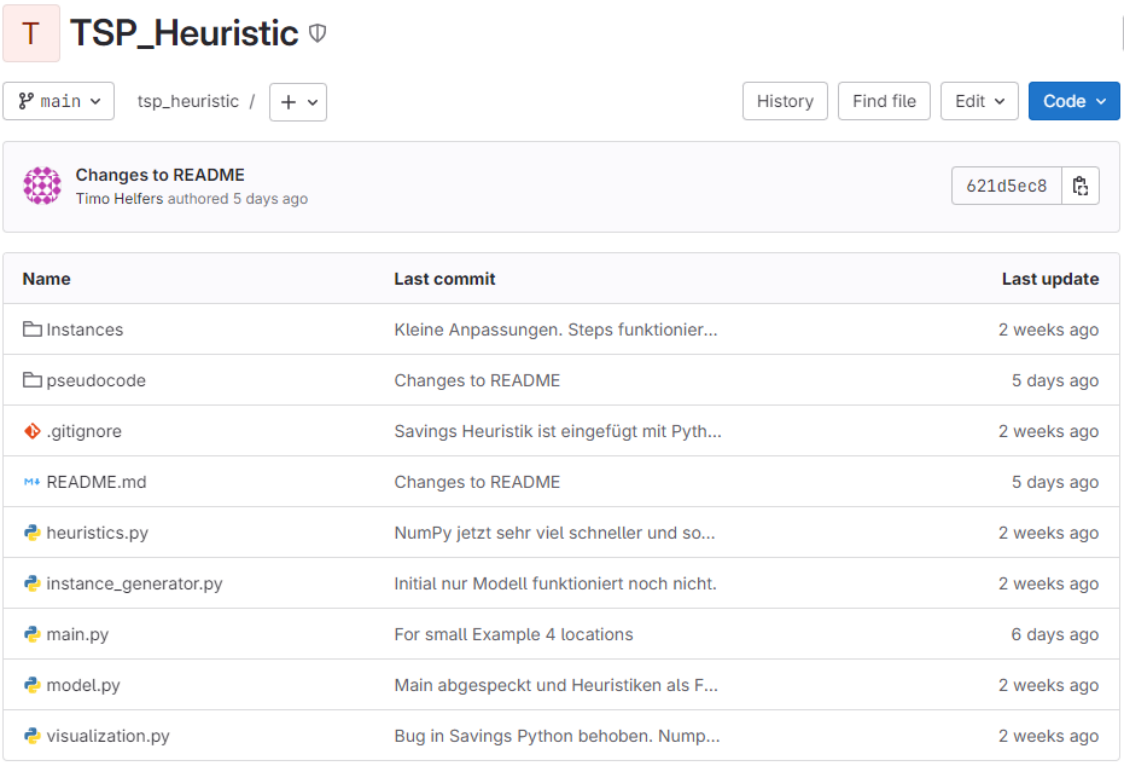
This way you should not be asked for your information every time you use a git function. If git does not work for you, feel free to contact us and we will search for a solution.

7.2 Repositories

7.2.1 What is a Repository?

A repository in Git is a project you or someone else is working on. In it we have the code-files of our program and other files needed to run our code, like instances, jpg’s etc. . You can also see the last commits in which a file was changed in any form. A commit means we change something in our code locally and want to save those changes for everybody. Every commit needs a commit message in which you have to describe what you have done. This makes it much easier to search for something in older commits for example when you want to change a part of code back to an old version.

In a repository you can have branches. A branch is an independent line of development. This means every branch is kind of a different version of your main code. You could for example have different branches for different model formulations. This way you can just switch between branches to run another formulation instead of rewriting your code every time. In this example we are on the main-branch, so every file is on the version of the last commit on this branch.



Usually we also have a `README.md` in it, which gives us informations about what the code does and how to use it. When creating the repository a `README.md`-file is generated automatically with a little summary on how to add files with git and other commands. To give others a better overview of the project you should edit the `README.md` accordingly. It is written in markdown in which we also wrote the texts, tables and model formulations of this lecture.

README.md

The Travelling Salesperson Problem

This code solves the TSP with three different solution methods, times the run time of each and returns it and the solutions in a visualization of the route.

Notation

Symbol	Description
Indices and Index Sets:	
$i, j \in \mathcal{N}$	Set of locations
Parameter:	
c_{ij}	Cost of travelling from location i to location j
Variables:	
$x_{ij} \in \{0, 1\}$	1 if travel from location i to location j, else 0
$z_i \in \mathbb{N}^+$	Integer variable for the order of the locations

Model Formulation

Objective Function:

1. $$\text{Min} \quad OFV = \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{N}} c_{ij} \cdot x_{ij}$$

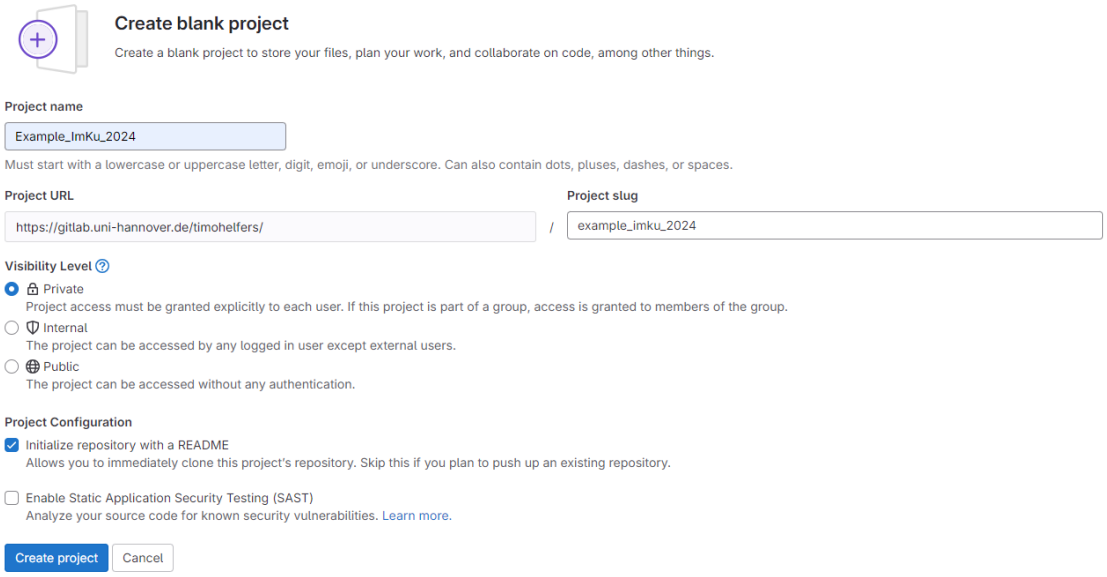
Constraints:

2. $$\sum_{j \in \mathcal{N}, j \neq i} x_{ij} = 1 \quad \forall i \in \mathcal{N}$$

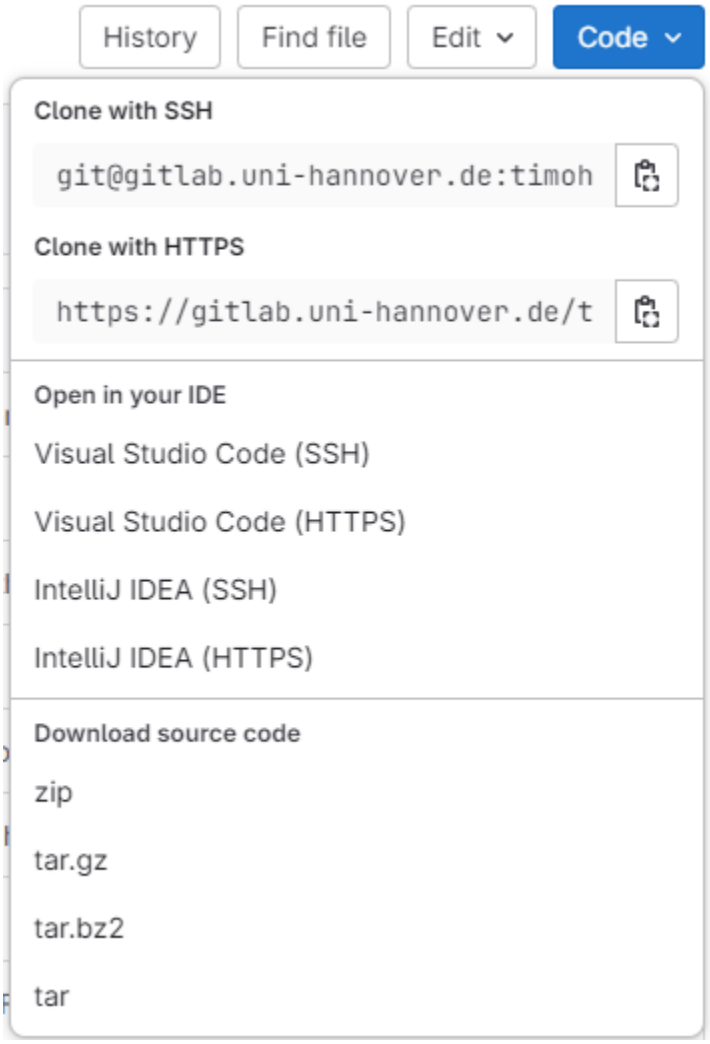
Additionally to the `README.md`, there is also a so called `.gitignore`-file in the repository. This file is not generated automatically and you need to add it manually after creating the repository. It is used to tell Git what we don't want to save in our repository. This sounds quit irritating at first, but usually we do not want to save everything that is in our folder on our PC. Examples for this are the Output-files of our code. Those get generated every time we run the code so there is no need to save them. As well as the `__pycache__`-files. Those also get generated when running the code and don't need to be saved. A typical `.gitignore`-file can look like this:

7.2.2 Create and Clone a Repository

Now that we have a basic understanding of what a repository is, we want to work with one. Firstly we will create a new repository. For that we navigate to our projects and click on the `New project` button in the top right corner. Now we select `Create blank project` and get to a page where we can decide on the project name and the visibility level.



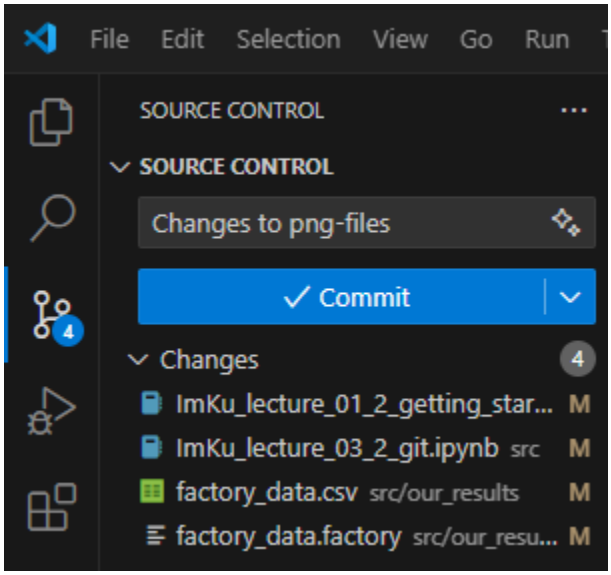
When everything is set we can create the project and get an empty repository only with the automatically generated `README.md`-file mentioned earlier. To work with this repository we need to clone it to our PC. This way we don't get a copy of the code but a link to the repository, so we can synchronize changes. To do this we click on the blue `Code`-Button in GitLab click on `Visual Studio Code (HTTPS)`.



Visual Studio Code and another window should open up in which you can navigate to the folder where you want to save the repository. After selecting a folder the repository gets cloned and we are able to access it in VS Code.

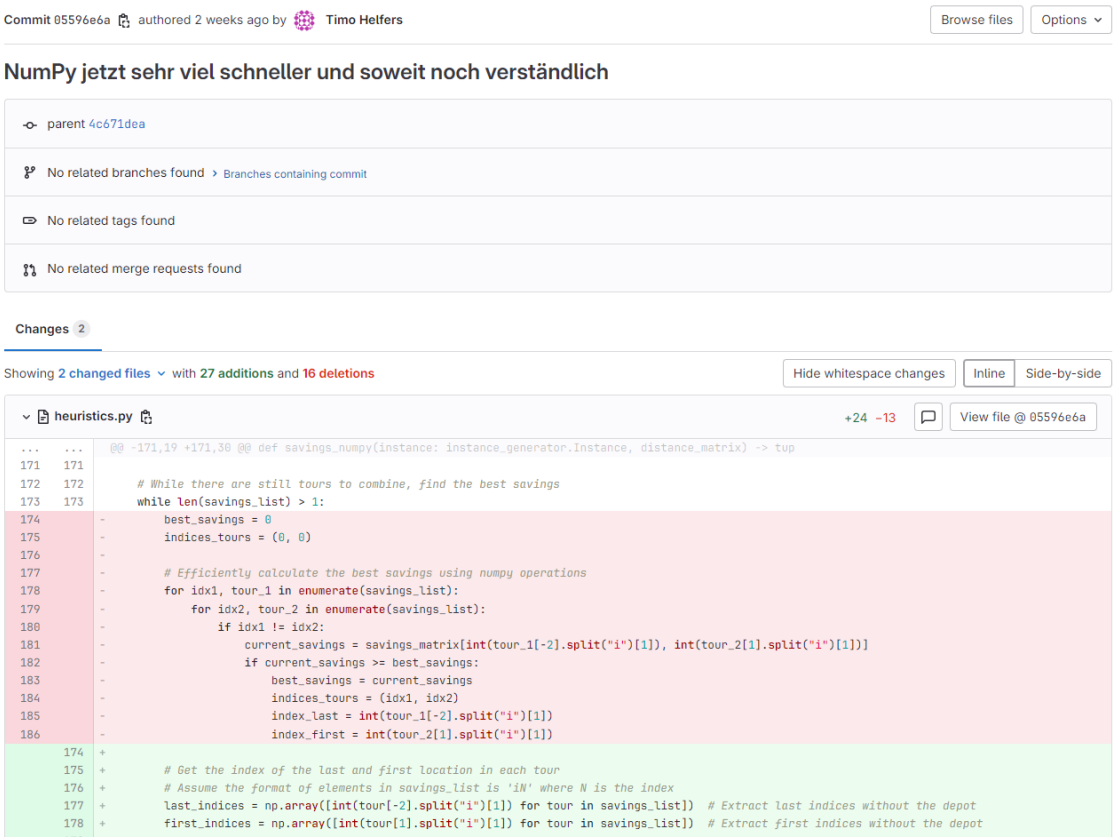
7.3 Working with Commits

As discussed earlier a commit saves the changes to the original-files in the repository. When you clone a repository to your PC you can change, add and delete every file in the repository. But before those changes are synchronized with the repository saved in GitLab you need to `push` a commit. This means you can decide if you want to save the changes made or just discard them as they are not working as expected. This way you can work on your code without the risk of breaking it.



If you work with others on a project they are also allowed to `push` commits. To get the changes they made you have to synchronize and `pull` them from GitLab. This way you work on the same version. You can synchronize your project by clicking on the arrow wheel in the bottom bar next to your branch name.

As mentioned before we can access earlier commits we made to for example change a part of your code back to an older version. For this we open the project in our browser in GitLab and click on the `History`-button. Now we can see every commit we've ever done in this project on this branch and by clicking on one of them we see the additions and deletions to the files colorized by green and red respectively.



By clicking on the button `Browse files` we get to all files in the branch on this commit without the deletions and additions. If we click on any of the files we will get the file as it was saved in this commit.

Chapter 8

Code Style and Documentation

8.1 PEP 8

Writing good code is not just about writing code that works, it is also about writing code that is easy to read and understand. PEP 8 (Python Enhancement Proposal) provides guidelines for writing clear and well-structured Python-code. You are not forced to follow this guideline, but we highly recommend it.

You can find the documentation of PEP 8 at https://peps.python.org/pep-0008/. In this chapter, we will give you an overview over the most important guidelines and our recommendations.

8.2 Naming Conventions

Naming things is very important, and sometimes it is really hard to find a good name for something. Nevertheless, there are a few rules that you should follow when naming things in Python.

The first isn't actually the names, but how you write them. They should follow a consistent style. When this style is not already specified by the project you are working on, you should follow the PEP 8 style guide. The essence of that is:

- Use `snake_case` for variable and function names.
- Use `PascalCase` for class names.
- Use `UPPER_SNAKE_CASE` for constants.
- Don not use single character variable names, except for counters in loops. (We sometimes break this rule when doing math.)
- Do not use cryptic abbreviations. It is fine if your variable names consist of multiple words, but do not make them too long either. If you have a very big expression, you can still use local short hands to make it a bit more compact.

Try to name your variables and functions so that it is obvious what the variable contains or how it is calculated, or what the function does without having to look up the definition.

8.3 Code Layout

Keep your code layout clean and consistent. You can use the VS Code Extension `Black Formatter`, which will help you to automatically format your code (but not every aspect, so you have to still check your code yourself.). Make sure that you activate the default formatter under settings and that formatting should be done automatically when saving.

To increase the readability of the code, *use blank lines* in selected places (two blank lines around top-level classes and functions, one line around functions in inside classes and one line around logically connected things.). Do not do:

```
def calculate_squares(numbers):
    squares = []
    for number in numbers:
        squares.append(number**2)
    return squares
def my_function():
    print("Hello from my function")
numbers = [1, 2, 3, 4, 5]
squares = calculate_squares(numbers)
for square in squares:
    print(square)
```

Instead do:

```
In [2]: def calculate_squares(numbers):
            squares = []
            for number in numbers:
                squares.append(number**2)
            return squares


        def my_function():
            print("Hello from my function")
```

```
In [3]: numbers = [1, 2, 3, 4, 5]
        squares = calculate_squares(numbers)

        for square in squares:
            print(square)
```

```
1
4
9
16
25
```

Only use 79 characters per line. Do not do:

```
# This is an example of a function with a long name, many arguments and a very long comment to show that it is useful to use linebreaks to make the code more readable than t

def this_is_a_long_function_name(variable_one, variable_two, variable_three, variable_four, variable_five, variable_six, variable_seven, variable_eight, variable_nine, varia
    pass
```

Instead do:

```
In [4]: # This is an example of a function with a long name, many arguments and a very
        # long comment to show that it is useful to use linebreaks to make the code
        # more readable than this long comment.


        def this_is_a_long_function_name(
            variable_one,
            variable_two,
            variable_three,
            variable_four,
            variable_five,
            variable_six,
            variable_seven,
            variable_eight,
```

```
            variable_nine,
            variable_ten,
        ):
            pass
```

Leave spaces around operators and break lines before the operator. Do not do:

```
i=23+324234+34545545+2909-123249990+3423423400+234324+34+67567+122-4354355+234
```

Instead do:

```
In [5]: i = (
            23
            + 324234
            + 34545545
            + 2909
            - 123249990
            + 3423423400
            + 234324
            + 34
            + 67567
            + 122
            - 4354355
            + 234
        )
```

Spaces: Do not leave spaces around brackets, commas etc. Leave one space around assignments. Do nor leave spaces around keyword arguments, but between different keyword arguments. Do not do:

```
my_dict = { 'animal' : 'dog', 'race' : [ 'German Shepherd' , 'Labrador Retriever' ] }

x=3

y               = 2
long_variable_name = 3

my_function(i = 3, j = 5)
my_function(i=3,j=5)
```

Instead do:

```
In [7]: def my_function(i, j): ...


        my_dict = {
            "animal": "dog",
            "race": ["German Shepherd", "Labrador Retriever"],
        }  # no space after the colon

        x = 3  # space around assignment

        y = 2  # one space around assignment
        long_variable_name = 3  # even though the sign does not align

        my_function(i=3, j=5)  # space between arguments, but not around equal sign
        my_function(i=3, j=5)
```

Have *imports* at the top with each import in a separat line. Separate different groups of modules by a blank line. Do not do:

```
def my_function(): ...


import sys, math
from os import path, walk

print("Hello, World!")
my_function()

import pandas as pd

import myownmodule
```

Instead do:

```
In [ ]: import sys  # standard library imports at the top
        import math  # each import in a new line
        from os import path, walk  # this is ok (as it is one module)

        import pandas as pd  # group with other third party imports

        # import myownmodule  # group with other local imports


        def my_function(): ...


        print("Hello, World!")
        my_function()
```

8.4 Comments

Make sure that your code is sufficiently commented. But do not comment every line or things that are obvious. You can use block comments, staring with a #and a singlie space. It is also possible to use inline comments. They should be separated by two spaces from the code.

```
In [ ]: # Block comments explain larger sections of code and provide a comprehensive
        # description of what the code does.
        #
        # For example, this block comment describes the following function in detail.
        #
        # We can create paragraphs by adding an empty line with a # between lines of␣
 ↪text.

        lower_bound = 0  # Set lower_bound to 0.
        upper_bound = 100  # Set upper_bound to 100.
        i = 0  # These are all examples of inline comments.

        # However, these comments are not necessary as the code is self-explanatory.
        # Better do:

        # Initializing variables
        lower_bound = 0
        upper_bound = 100
        i = 0
```

Use docstrings and type hints to create a good documentation and to help your IDE give you better autocomplete suggestions (see section 5.7).

```
In [ ]: def documentation_example(i: int) -> str:
            """
            This function takes an integer as input, prints the sum of the integer with␣
 ↪itself,
            and returns the string "hello".

            Args:
                i (int): The integer to be processed.
            Returns:
                str: The string "hello".
            """
            print(i + i)
            return "hello"
```

8.5 Use More Functions

Functions are very powerful, use them! Once you find yourself writing the same code twice, you should think about writing a function for it. If you write it three times, you should have already written the function. Often, it will also make sense to write a function even if you only use it once. You can do this to make it clear that this is a meaningful sub-operation of your function. This will also prevent your functions from turning into a 100-line mess.

8.6 Readability is King

Whenever you are unsure about how to write something, go for the more readable option. You should always strive to write the least complex code possible. since complexity is a fairly good measure of how hard it is to understand something.

```
In [9]: # difficult to understand
        print(list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, range(20)))))
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

```
In [13]: # better
         def filter_even_numbers_and_square(numbers):
             return [number**2 for number in numbers if number % 2 == 0]


         numbers = list(range(20))
         print(filter_even_numbers_and_square(numbers))
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

```
In [14]: # too detailed
         def filter_even_numbers(numbers):
             filtered_numbers = []
```

```
            for number in numbers:
                if number % 2 == 0:
                    filtered_numbers.append(number)
            return filtered_numbers


        def square_numbers(numbers):
            squared_numbers = []
            for number in numbers:
                squared_numbers.append(number**2)
            return squared_numbers


        numbers = list(range(20))
        even_numbers = filter_even_numbers(numbers)
        squared_numbers = square_numbers(even_numbers)
        print(squared_numbers)
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

8.7 Don't Be Afraid to Burn It All Down

It is highly unlikely that you will get everything right the first time. Burning it all down and starting from scratch is often way less work than you think, since most of the work was probably trying to figure out the problem, not writing the code. You can carry over that experience to the new code and write it much faster and better without the holes that you dug yourself into in the first attempt.

Chapter 9

Object-Oriented Programming

9.1 Basics

In this lecture, we will give a very brief introduction to object-oriented programming (OOP). As a multi-paradigm language, Python also offers object orientation.

What is object-oriented programming? The basic idea is to put data and the functions that act upon that data in one place: an object. The data members and functions are specified in that object's class. The class can be thought of as the blueprint for the objects.

The simplest kind of objects are just data containers with name fields. A class is defined with the `class` keyword followed by the class name. Let's create a very simple class for students. We can have default values, as the `semester` in this case.

```
In [21]: class Student:
             name = None
             age = None
             semester = 1


         student_1 = Student()
         student_1.name = "Clara"
         student_1.age = 21

         student_2 = Student()
         student_2.name = "Max"
         student_2.age = 22
         student_2.semester = 3

         print(
             f"{student_1.name} is {student_1.age} years old and in semester {student_1.
 ↪semester}."
         )
         print(
             f"{student_2.name} is {student_2.age} years old and in semester {student_2.
 ↪semester}."
         )
```

```
Clara is 21 years old and in semester 1.
Max is 22 years old and in semester 3.
```

9.2 Methods

9.2.1 `__init__()`

In Python, all classes can have a special function called `__init__()`. When creating an instance of a class, this function is automatically executed. It is used to initialize the object's attributes or to perform any setup procedures required when the object is created. The initializer should not return a value, as the creation and returning of the object itself is handled by Python internally.

Instance methods always take `self` as their first parameter. The `self` parameter is a reference to the current instance of the class, allowing access to its attributes and methods. (Note: `self` is just a variable name. Theoretically, you could use any other name. However, it is a strong convention in Python to call it `self`.)

We now use the `__init__()` function to set the attributes of our students. We use again a default value for the `semester`.

```
In [22]: class Student:
             def __init__(self, name, age, semester=1):  # default value
                 self.name = name
                 self.age = age
                 self.semester = semester


         student_1 = Student("Clara", 21)  # create a new student
         student_2 = Student("Max", 22, 3)

         print(
             f"{student_1.name} is {student_1.age} years old and in semester {student_1.
 ↪semester}."
         )
         print(
             f"{student_2.name} is {student_2.age} years old and in semester {student_2.
 ↪semester}."
         )
```

```
Clara is 21 years old and in semester 1.
Max is 22 years old and in semester 3.
```

9.2.2 Self-defined Methods

We can also define our own function within the class. Note that when we define such a function inside a class, it is called a method. This is because we can access it only through an instance of the class (or the class itself), not independently in the global scope of our code.

Let's define a method that introduces the student.

```
In [23]: class Student:
             def __init__(self, name, age, semester=1):
                 self.name = name
                 self.age = age
                 self.semester = semester

             def introduce(self):
                 return (
                     f"I am {self.name}, {self.age} years old and in semester {self.
 ↪semester}."
```

```
                )



        student_1 = Student("Clara", 21)
        print(student_1.introduce())
```

```
I am Clara, 21 years old and in semester 1.
```

Not all attributes of an object have to be initialized directly when the object is created (in the `__init__()` constructor). Instead, some attributes can be added or changed later in the life cycle of the object. This provides flexibility as additional information or states can be managed only when needed.

In our example, we could create a student at the beginning of its studies. We know that the student will (later) have grades, so we create an empty dictionary called `grades`. As the student progresses, grades can be added to the student using the `add_grade()` function.

```
In [24]: class Student:
             def __init__(self, name, age, semester=1):
                 self.name = name
                 self.age = age
                 self.semester = semester
                 self.grades = {}

             def introduce(self):
                 return (
                     f"I am {self.name}, {self.age} years old and in semester {self.
 ↪semester}."
                 )

             def add_grades(self, grades: dict):
                 """
                 Adds or updates the grades for the student.

                 Parameters:
                 grades (dict): A dictionary containing subject names
                 as keys and corresponding grades as values.

                 Returns:
                 None
                 """
                 self.grades.update(grades)  # update the dict



         student_1 = Student("Clara", 21)
         print(student_1.introduce())
         student_1.add_grades({"BWL I": 1.3, "BWL II": 1.3, "VWL I": 3.0, "Math I": 2.0})

         print(student_1.grades)
```

```
I am Clara, 21 years old and in semester 1.
{'BWL I': 1.3, 'BWL II': 1.3, 'VWL I': 3.0, 'Math I': 2.0}
```

9.2.3 Printing Objects

In addition to `__init__()`, classes can have other special methods. Above, we wrote our own function for introducing the student. For displaying an object, we can also use the special methods `__repr__()` (short for representation) and `__str__()`. These methods are called when something like the `print()` function wants to convert our object into a string.

- `__str__()`: Representing the string in a readable format, particularly for users.
- `__repr__()`: Representing the string in a way that can be used to recreate the object (machine perspective).

```
In [25]: class Student:
             def __init__(self, name, age, semester=1):
                 self.name = name
                 self.age = age
                 self.semester = semester

             def __repr__(self):
                 return (
                     f"Student(name='{self.name}', age='{self.age}', semester='{self.
↪semester}')"
                 )

             def __str__(self):
                 return f"{self.name}: {self.age} years old and in semester {self.
↪semester}"



         student_1 = Student("Clara", 21)
         print(student_1)  # uses __str__, if __str__ is not implemented, __repr__ would␣
↪be used
         print(str(student_1))
         print(repr(student_1))  # equivalent to print(student_1.__repr__())
```

```
Clara: 21 years old and in semester 1
Clara: 21 years old and in semester 1
Student(name='Clara', age='21', semester='1')
```

9.2.4 Operator overloading

Say we want to check if to students are equal (*Let's define equal as the same name, age and semester. In reality, this would not work so easily. But let this assumption be sufficient as a teaching example.*).

We can write a function for checking the equality of a student (`self`) and another student (`other`), returning `True` if the students are equal:

```
In [26]: class Student:
             def __init__(self, name, age, semester=1):
                 self.name = name
                 self.age = age
                 self.semester = semester

             def __repr__(self):
```

```
                return (
                    f"Student(name='{self.name}', age='{self.age}', semester='{self.
↪semester}')"
                )

            def __str__(self):
                return f"{self.name}: {self.age} years old and in semester {self.
↪semester}"

            def check_equal(self, other: "Student") -> bool:
                """
                Compares the current Student object with another Student object to␣
↪check for equality.

                Args:
                    other (Student): The other Student object to compare with.
                Returns:
                    bool: True if both Student objects have the same name, age, and␣
↪semester; False otherwise.
                """

                return (
                    self.name == other.name
                    and self.age == other.age
                    and self.semester == other.semester
                )
```

Create three students and check if they are equal using the function.

```
In [27]: student_1 = Student("Clara", 21)
         student_2 = Student("Hannes", 22)
         student_3 = Student("Clara", 21)

         print(student_1.check_equal(student_2))
         print(student_1.check_equal(student_3))
```

```
False
True
```

We can do the same using a *magic method*. *Magic methods* are special methods that begin and end with double underscores, e.g., `__add__()` or `__eq__()`. These methods allow us to define the behavior of built-in operations for our own objects. For instance, `__eq__()` allows us to define how the `==` operator works with our objects and `__add__()` would define the `+` operator. This change in the behavior of an existing operator is called *operator overloading*.

In our case, we use the `==` operator to to check for equality of two students. Thus, we define the `__eq__()` method. Similar to our own function, it takes two arguments (`self` and `other`) and checks equality. However, instead of calling `student_1.__eq__(student_2)`, we can use the `==` operator, which leads to the simple line `student_1 == student_2`.

```
In [28]: class Student:
             def __init__(self, name, age, semester=1):
                 self.name = name
```

```
                self.age = age
                self.semester = semester

            def __repr__(self):
                return (
                    f"Student(name='{self.name}', age='{self.age}', semester='{self.
↪semester}')"
                )

            def __str__(self):
                return f"{self.name}: {self.age} years old and in semester {self.
↪semester}"

            def __eq__(self, other: "Student") -> bool:
                """
                Check if two Student objects are equal.

                Args:
                    other (Student): The other Student object to compare with.
                Returns:
                    bool: True if both Student objects have the same name, age, and␣
↪semester, False otherwise.
                """

                return (
                    self.name == other.name
                    and self.age == other.age
                    and self.semester == other.semester
                )
```

```
In [29]: student_1 = Student("Clara", 21)
         student_2 = Student("Hannes", 22)
         student_3 = Student("Clara", 21)

         print(student_1 == student_2)
         print(student_1 == student_3)
```

```
False
True
```

Operator overloading can also be done for other operators: - `+` (Addition): `__add__(self, other)` - `-` (Subtraction): `__sub__(self, other)` - `*` (Multiplication): `__mul__(self, other)` - `/` (Division): `__truediv__(self, other)` - `%` (Modulo): `__mod__(self, other)` - `**` (Power): `__pow__(self, other)` - ...

Using operator overloading makes code more intuitive and readable. It allows us to use natural arithmetic operators to perform operations on user-defined objects, making complex operations simpler to write and understand.

9.3 Inheritance

In many cases, classes share a lot of similarities, but also have distinct properties that set them apart. For instance, consider the case of students. Bachelor's and Master's students share many common characteristics,

but they also differ in some details. One approach would be to create two separate, independent classes for each type of student. However, this would mean duplicating the shared properties in both classes, leading to redundant code.

Inheritance offers a more efficient solution. By using inheritance, we can create a base class that contains the shared properties, from which other classes (child classes) can be derived. This way, the shared properties are defined only once in the base class, and the derived classes can inherit these properties while adding their unique attributes.

In our example, the base class could be `Student`. It has some properties as well as a representation function and a function for adding grades:

```
In [30]: # Base class (as before)
         class Student:

             def __init__(self, name, age, semester=1):
                 self.name = name
                 self.age = age
                 self.semester = semester
                 self.grades = {}

             def __str__(self):
                 return f"{self.name}: {self.age} years old and in semester {self.
 ↪semester}"

             def add_grades(self, grades: dict):
                 self.grades.update(grades)
```

```
In [31]: general_student_clara = Student("Clara", 21)
         print(general_student_clara)
```

```
Clara: 21 years old and in semester 1
```

Based on this class, we can now define a Bachelor student. In addition to the student attributes, each student will have a Bachelor thesis. In addition, we assume that each Bachelor student should have completed a pre-internship before starting their studies (so the default value is `True`). With `super()`, we can access the methods and attributes of the base class. Thus, `super().__init__(name, city)` will call the constructor from the base class to initialize the object's attributes.

For our string representation, we want to add some new information to the existing description from the base class. Thus, we get the description from the base class with `super().__str__()` and add our new information. As we did in other classes, we can define a specific function solely for this class.

```
In [32]: class BachelorStudent(Student):

             def __init__(self, name, age, semester=1, bachelor_thesis=None,␣
 ↪internship=True):
                 super().__init__(name, age, semester)
                 self.bachelor_thesis = bachelor_thesis
                 self.internship = internship
                 # grades are inherited from the base class

             def __str__(self):
                 base_description = super().__str__()
                 return f"{base_description}, Bachelor, Thesis: {self.bachelor_thesis}"
```

```
            def set_bachelor_thesis(self, thesis):
                self.bachelor_thesis = thesis
```

Let's create a bachelor student.

```
In [33]: bachelor_student_anna = BachelorStudent("Anna", 21)
         print(bachelor_student_anna)
         print(bachelor_student_anna.internship)
```

```
Anna: 21 years old and in semester 1, Bachelor, Thesis: None
True
```

We can use the method `set_bachelor_thesis` to set the title of the thesis as we used other methods before.

```
In [34]: bachelor_student_anna.set_bachelor_thesis("The impact of KI on society")
         print(bachelor_student_anna)
```

```
Anna: 21 years old and in semester 1, Bachelor, Thesis: The impact of KI on society
```

We can also use the method `add_grades()`, which is inherited from the base class `Student`.

```
In [35]: bachelor_student_anna.add_grades(
             {"BWL I": 1.3, "BWL II": 1.3, "VWL I": 3.0, "Math I": 2.0}
         )
         print(bachelor_student_anna.grades)
```

```
{'BWL I': 1.3, 'BWL II': 1.3, 'VWL I': 3.0, 'Math I': 2.0}
```

As an exercise: Create another child class for master students. The master students should have a major and a representation. In addition, there should be a special award for all students whose grade is below 1.5. Create a function within the class that returns whether a master student gets this award or not.

```
In [36]: class MasterStudent(Student):
             def __init__(self, name, age, major, semester=1):
                 super().__init__(name, age, semester)
                 self.major = major

             def __str__(self):
                 base_description = super().__str__()
                 return f"{base_description}, Master student with major " f"{self.
 ↪major}'."

             # check whether the mean grades of the student is below 1.5
             def very_good_student(self):
                 if len(self.grades) == 0:
                     print(f"The student {self.name} has no grades yet.")
                 elif sum(self.grades.values()) / len(self.grades) < 1.5:
```

```
                print(f"The student {self.name} gets the award.")
            else:
                print(f"The student {self.name} does not get the award.")
            return


    # Create a master student
    master_student_1 = MasterStudent(name="Jan", age=25, semester=3, major="OR")
    print(master_student_1)
    master_student_1.very_good_student()

    master_student_1.add_grades({"Logistik": 1.0, "OR": 1.3, "ProdProz": 1.7})
    master_student_1.very_good_student()
```

```
Jan: 25 years old and in semester 3, Master student with major OR'.
The student Jan has no grades yet.
The student Jan gets the award.
```

9.4 Instance Data based on Classes

Where are classes used in our area? They are very useful when data has to be generated, especially instance data for models. We can build the structure of an instance once and then generate instances very easily. In the following example, cities and their coordinates are generated based on a number of cities. This allows different instances to be generated (very easily) so that different instances can be analyzed with models.

```
In [37]: import random
         import math


         class MyInstance:
             """
             MyInstance is a class that represents a Traveling Salesman Problem (TSP)␣
↪instance.

             Attributes:
                 nb_locations (int): The number of locations.
                 locations (list): A list of location identifiers.
                 instance_name (str): The name of the TSP instance.
                 coordinates (dict): A dictionary mapping location identifiers to their␣
↪coordinates.
                 distance (dict): A dictionary containing the pairwise distances between␣
↪locations.

             Methods:
                 calculate_distance() -> dict:
                     Calculates the distance between all cities.
             """

             def __init__(self, nb_locations: int):
                 """
                 Initialize the TSP instance with a given number of locations.
```

```
                Args:
                    nb_locations (int): The number of locations to generate.

                """
                random.seed(42)
                self.nb_locations = nb_locations
                self.locations = [f"i{i}" for i in range(1, nb_locations + 1)]
                self.instance_name = f"tsp_i{self.nb_locations}"
                self.coordinates = {
                    i: (round(random.uniform(0, 50), 2), round(random.uniform(0, 50),␣
↪2))
                    for i in self.locations
                }

                self.distance = self.calculate_distance()

            def __str__(self) -> str:
                """
                Returns a string representation of the instance.

                Returns:
                    str: A formatted string containing the instance name and number of␣
↪locations.
                """
                return f"Instance: {self.instance_name}, Locations: {self.nb_locations}"

            def calculate_distance(self) -> dict:
                """
                Calculate the pairwise Euclidean distance between all locations.

                This method iterates over all pairs of locations and computes the␣
↪Euclidean
                distance between them. The distances are stored in the `self.distance`␣
↪dictionary
                with the tuple of location pairs as keys.

                The distance from a location to itself is set to 0.

                Returns:
                    dict: A dictionary containing the pairwise distances between␣
↪locations.
                """
                distance = {}

                for i, location_1 in enumerate(self.locations):

                    # the travel time to the location itself is 0
                    distance[(location_1, location_1)] = 0

                    # select the next location
                    for location_2 in self.locations[i:]:

                        x1, y1 = self.coordinates[location_1]
                        x2, y2 = self.coordinates[location_2]
```

```
                        distance[(location_1, location_2)] = math.sqrt(
                            (x1 - x2) ** 2 + (y1 - y2) ** 2
                        )
                        distance[(location_2, location_1)] = distance[(location_1,␣
↪location_2)]
                return distance


        my_instance = MyInstance(5)
        print(my_instance)
        print(my_instance.coordinates)
```

```
Instance: tsp_i5, Locations: 5
{'i1': (31.97, 1.25), 'i2': (13.75, 11.16), 'i3': (36.82, 33.83), 'i4': (44.61, 4.35),␣
↪'i5': (21.1, 1.49)}
```

If we now generate this data, we can simply save it with pickle. Later, these instances (together with the class) can then be reloaded and used in another file.

```
In [38]: import os
         import pickle

         # create a folder
         path = "../data/city_data"
         if not os.path.exists(path):
             os.makedirs(path)

         with open(f"{path}/my_instance.pkl", "wb") as f:
             pickle.dump(my_instance, f)
```

And we can even generate multiple instances at once:

```
In [39]: for nb_city in range(3, 10):
             with open(f"../data/city_data/instance_{nb_city}.pkl", "wb") as f:
                 instance = MyInstance(nb_city)
                 pickle.dump(instance, f)
```

9.5 Documenting Classes

As for functions, you should use type hints and docstrings for documenting classes.

```
In [40]: class Student:
             """
             A class to represent a student.

             Attributes
             ----------
             name : str
                 The name of the student.
             age : int
```

```
            The age of the student.
        semester : int, optional
            The current semester (default is "1").

        Methods
        -------
        introduce():
            Returns a string introducing the student.
        """

        def __init__(self, name: str, age: int, semester=1):
            """
            Initializes a new instance of the class.

            Args:
                name (str): The name of the person.
                age (int): The age of the person.
                semester (int, optional): The current semester. Defaults to "1".
            """
            self.name = name
            self.age = age
            self.semester = semester

        def introduce(self) -> str:
            """
            Introduce the person with their name, age, and semester.

            Returns:
                str: A string containing the person's name, age, and semester.
            """
            return (
                f"I am {self.name}, {self.age} years old and in semester {self.
↪semester}."
            )
```

Chapter 10

Python Scripts

10.1 Python Scripts vs. Jupyter Notebooks

While Jupyter Notebooks are a very convenient tool for interactive Python, they are not the only or original way of executing Python. The normal way is to use Python scripts, which is useful for writing large programs or libraries.

First things first: code works in both, Python and Jupyter Notebook. The difference lies more in how the code is executed and how results are displayed.

Jupyter Notebooks are particularly interactive. We can split our code into different cells and run them one after the other. This is especially useful for teaching examples, such as in this course. It is also very helpful for data evaluation: as you have already noticed, the things that have already been executed are remembered (until the kernel is restarted). This means, for example, that data can be loaded first and then be evaluated step by step. The outputs are also displayed very clearly and can, in some cases, be used interactively. Jupyter Notebooks are therefore very well-suited for setting up or testing things step by step and are excellent for learning and teaching.

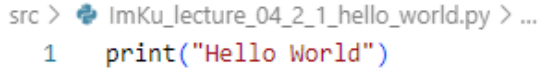
In comparison, a Python script is executed in its entirety. The results are printed in the console. While you can print intermediate results, you cannot change your program while it is running (in contrast to a Jupyter Notebook, where you can just add another cell once the previous cell is executed). Python scripts can be easily used to create modular and reusable code. You might have noticed (and you will see) that in some lectures, the Jupyter Notebooks can be very long. With Python scripts, we can outsource functions into other files and still incorporate these functions into our main code. This leads to a very modular design and the ability to incorporate these functions into different scripts. In this way, we define our own modules and use them, just like other modules (see Chapter 3 for comparison).

Overall, the choice between Jupyter Notebooks and Python scripts strongly depends on the application.

10.2 Basic Python Scripts

Let's create a simple file. The file has to end with `.py` to declare it as a Python script. Write something in the file like the following:

```
print("Hello World")
```



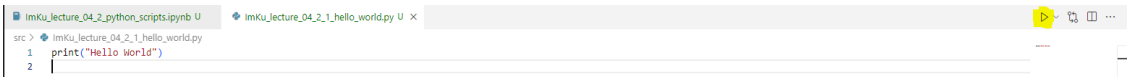
We can execute the file over the terminal: You can open a terminal In VS Code with `Terminal -> New Terminal`. Navigate to the specific folder and execute the file using `python3 filename.py`:

```
root@0637f697ab3b:/workspaces/imku2023# cd src
root@0637f697ab3b:/workspaces/imku2023/src# python3 ImKu_lecture_04_2_1_hello_world.py
Hello World
root@0637f697ab3b:/workspaces/imku2023/src#
```

You can do the same in a Windows terminal: search for `cmd`, than navigate to the current folder `src` and the run `python3 helloworld.py`:

```
\imku2023>cd src

\imku2023\src>python3 ImKu_lecture_04_2_1_hello_world.py
```

Of course, we can also execute the file in VS Code directly using the arrow at the top right corner.



10.3 Scripts as Modules

Let's change our Python script a bit. Instead of just printing `Hello World` create a function:

```
def say_hello():
    print("Hello from a function")
```

We can now import the script in **another Python file or in this Jupyter Notebook**, just as we did it with other modules.

```
In [1]: from ImKu_lecture_04_2_1_hello_world import say_hello

        say_hello()

        # or:
        # import ImKu_lecture_04_2_1_hello_world

        # ImKu_lecture_04_2_1_hello_world.say_hello()
```

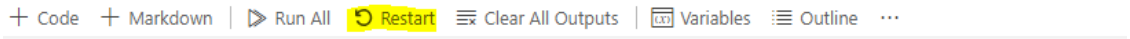
```
Hello from a function
I am a function
```

Let's add any other print statement `I am a function` to the function in the Python script and run the cell below.

```
In [2]: # from hello_world import say_hello  # still not working with an additional␣
 ↪import

        say_hello()  # only prints the statement from before
```

```
Hello from a function
I am a function
```

You will see that only the previous print statement is executed, not the one you just added. Why is this the case? In a Jupyter Notebook, imported modules are loaded into memory. If the Python file is changed, it is not automatically re-imported. Even if you re-run the import statement, Python does not reload the modified file automatically since it is already in the internal cache and assumes it has not changed. To run the updated Python file, **you must restart the kernel** and run your entire Jupyter Notebook again



After the restart, it works (see output above).

10.4 `main()`

Let's create a new Python file `calculate.py` with the following function:

```
def sum(i, j):
    return i + j


i = 1
j = 2
print(sum(i, j))
```

Run the following cell:

```
In [3]: from ImKu_lecture_04_2_2_calculate import sum

        print(sum(4, 5))
```

```
9
```

What do you notice?

We did not only calculate the sum of 4 and 5, as we intended. The sum of 1 and 2 from the Python file is also executed, even though we only imported the function `sum`. The reason for this is that the entire Python file is executed when it is imported, which results in the execution of the line `print(sum(i, j))`, leading to an output of 3.

To prevent this automatic execution, we should always put all processing code inside a function. However, simply putting our `print(sum(i, j))` statement inside any function won't work because that function wouldn't be called, and hence, we won't be able to execute the Python file directly.

The best practice in this case is to use the `main()` function pattern in our Python file:

```
def sum(i, j):
    return i + j


def main():
    i = 1
    j = 2
    print(sum(i, j))


if __name__ == "__main__":
    main()
```

`__name__ == "__main__"` checks whether the Python file is being run directly or imported as a module. If the file is being run directly, the `main()` function is executed. In our case, this means the lines that calculate the sum of 1 and 2 will be run, producing the output. If the file is not executed directly, for example, if it is imported into another file, the `main()` function is not executed. This ensures that no code is executed unintentionally when the file is imported, preventing any unwanted output or side effects.

By following this practice, we can ensure our code behaves as expected whether run directly or imported into another module

Remember: When changing the Python script, restart the kernel!

10.5 Paths in Python Scripts and Jupyter Notebooks

Create the following Python file checking the current working directory:

```
import os


def main():
    print_working_dir()


def print_working_dir():
    print(os.getcwd())


if __name__ == "__main__":
    main()
```

Let's import the function `print_working_dir()` and print the current working directory.

```
In [4]: from ImKu_lecture_04_2_3_working_dir import print_working_dir

        print_working_dir()
```

```
/workspaces/imku2023/src
```

Let's print the current working directory directly in this Jupyter Notebook.

```
In [5]: import os

        print(os.getcwd())
```

```
/workspaces/imku2023/src
```

And now, go to the Python file and execute it directly.

You will notice that the working directory is not `/workspaces/imku2023/src`, but `/workspaces/imku2023`! This will probably not be a problem for this course, as you will either import your Python scripts into Jupyter notebooks or work solely with Python files (and not execute a Python file directly and import the same file). However, you should still be aware of this difference. There are various ways to handle this issue, such as changing the working directory of the Python script with `os.chdir('/workspaces/imku2023/src')`.

10.6 Usage of Python Files and Jupyter Notebooks

When your project grows more complex, you will want to split your code into multiple files. This is also a nice way to keep your Jupyter Notebooks clean. If you have functions or classes that are very complicated, but it's not important for the reader to see them, you can put them in another file and import them.

Of course, multiple Python files can be loaded as modules. If a specific format (e.g., pure Python files or Jupyter Notebooks) is required for the exams, we will specify this explicitly. Otherwise, the principle is that you must find a suitable structure for your code which will of course also be included in the grading. Later on, e.g. in a Master thesis, you will probably mostly work with Python files (even though Jupyter Notebooks are also there very useful for numerical analysis).

Chapter 11

NumPy

11.1 General Information

NumPy is one of the most important libraries for Python. Basically, a library is a collection of functions that are not part of the programming language, but can be accessed by importing the library. So instead of implementing a function yourself, you should rather use a library that has already done it. This saves time and causes far fewer errors.

NumPy is mainly used to create list-like objects called arrays. These can be one-dimensional, two-dimensional, and so on. Compared to normal lists, using NumPy arrays gives you faster running times and other advantages, which we will learn about in this section. To illustrate the shorter runtime, we will use a small example.

```
In [42]: import numpy as np
         import timeit
         from time import perf_counter
```

We want to calculate the mean of a list of integers. This means we have to iterate through every element in the list, which can be very time-consuming, especially for large lists.

To make matters worse, we want to do this 100 times in a row. This could be the case for a heuristic with 100 iterations, where one step within an iteration is the calculation of the mean.

```
In [43]: # Creating a list and a numpy array
         numbers = list(range(1, 10000001))
         numbers_array = np.array(numbers)


         # Calculating the mean of the list
         def list_mean(numbers: list) -> float:
             return sum(numbers) / len(numbers)


         # Calculating the mean of the numpy array
         def array_mean(numbers_array: np.array) -> float:
             return np.mean(numbers_array)


         if list_mean(numbers) == array_mean(numbers_array):
             print(f"Both functions return the same mean value:␣
 ↪{array_mean(numbers_array)}")
```

```
        # Calculate the needed time to execute the functions 100 times
        list_time = timeit.timeit(lambda: list_mean(numbers), number=100)
        print(f"List time: {list_time} seconds")

        array_time = timeit.timeit(lambda: array_mean(numbers_array), number=100)
        print(f"Array time: {array_time} seconds")
```

```
Both functions return the same mean value: 5000000.5
List time: 6.074021514001288 seconds
Array time: 0.9294465450002463 seconds
```

As you can see, even for such trivial purposes as calculating the mean of a list, we get much faster runtimes with the NumPy array. Imagine how much faster a heuristic that does over 1000 iterations of the same calculations on large lists would be if we used NumPy arrays instead.

There are a few reasons why using NumPy arrays is often faster than using lists. First, an array contains only one type of data. This allows a contiguous block of memory to be allocated for an entire array. This one block can be accessed much faster than accessing fragments of a list scattered throughout memory. Secondly, because every element in an array is of the same type, the elements don't need to be type-checked during operations, saving runtime. Thirdly, the biggest contributor to faster runtimes is vectorisation. We can perform operations on entire arrays at once, rather than iterating through each element one at a time.

11.2 Initialise a NumPy-Array

11.2.1 One-Dimensional Arrays

To Initialise a NumPy-Array we can use a list.

```
In [44]: print(np.array([1, 2, 3, 4, 5]))
```

```
[1 2 3 4 5]
```

Another method is to use the built-in function `np.linspace()`. This function takes three inputs: `start`, `stop` and `count`. The first two inputs declare the range in which we want the values. The third input is the number of values we want to take from that range. The values are spread evenly over the range, with the same increment between all values.

```
In [45]: print(np.linspace(1, 10, 10))  # start, stop, count
         print(np.linspace(0.5, 25))  # the default count is 50
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[ 0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5  7.
  7.5  8.   8.5  9.   9.5 10.  10.5 11.  11.5 12.  12.5 13.  13.5 14.
 14.5 15.  15.5 16.  16.5 17.  17.5 18.  18.5 19.  19.5 20.  20.5 21.
 21.5 22.  22.5 23.  23.5 24.  24.5 25. ]
```

Similar to `np.linspace()` is `np.arange()`, but instead of specifying how many values we want to get, we specify the increment between the values in a given range. The default increment is 1.

```
In [46]: print(np.arange(1, 10, 1))
         print(np.arange(0.5, 25, 0.5))
```

```
[1 2 3 4 5 6 7 8 9]
[ 0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5  7.
  7.5  8.   8.5  9.   9.5 10.  10.5 11.  11.5 12.  12.5 13.  13.5 14.
 14.5 15.  15.5 16.  16.5 17.  17.5 18.  18.5 19.  19.5 20.  20.5 21.
 21.5 22.  22.5 23.  23.5 24.  24.5]
```

Sometimes we just need arrays containing only zeros or maybe even ones. This can be achieved with the functions `np.zeros()` and `np.ones()` respectively.

```
In [47]: print(np.zeros(5))
         print(np.ones(10))
```

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

11.2.2 Two-Dimensional Arrays

For the 2-dimensional case, we simply add a second input parameter. The first gives us the number of rows and the second the number of columns.

```
In [48]: print(np.zeros((5, 8)))
```

```
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

To initialise a two-dimensional array with evenly spaced values, we can again use the `np.arange()` function in combination with `reshape()`. This first creates a one-dimensional array with the specified values. It then reshapes it into the desired shape.

```
In [49]: two_d_array = np.arange(1, 16).reshape(3, 5)
         print(two_d_array)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
```

Another useful function for initialising arrays is `np.meshgrid()`. This function creates a grid with the same array for either the columns or the rows (for 2-dimensional arrays). This way we can create arrays with their row or column indices as values. In this case it takes two array-like structures as input.

```
In [50]: index_row, index_col = np.meshgrid(
             np.arange(0, 10), np.arange(0, 8), indexing="ij"
         )  # Use ij indexing for matrix-like indexing

         print(index_row)
         print(index_col)
```

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

11.2.3 Randomly Generated Arrays

It is also possible to initialise an array with random values in a specific shape. If we want random values of type `float`, we can use `np.random.random()`. This function generates random values from a uniform distribution between 0 and 1. To scale this up, we can easily multiply all values by a certain value and get distributions between, for example, 0 and 10.

To get random values of type `int` we can use the function `np.random.randint()`. This function takes the range for the random values and the desired shape of the matrix.

```
In [51]: random_array_dec = np.random.random((3, 3)) * 10  # 3x3 matrix

         random_array_int = np.random.randint(0, 11, (5, 3))  # 5x3 matrix

         print(random_array_dec)
         print(random_array_int)
```

```
[[0.58083612 8.66176146 6.01115012]
 [7.08072578 0.20584494 9.69909852]
 [8.32442641 2.12339111 1.81824967]]
[[ 4  0  9]
 [ 5  8  0]
 [10 10  9]
 [ 2  6  3]
 [ 8  2  4]]
```

One problem with randomly generated arrays is that we can't really compare our results if they change every time we run our code. To avoid this, we define a random seed from which the values are generated. This way we get random values, but they are the same every time we run our code.

```
In [52]: np.random.seed(42)
         random_array_seed = np.random.randint(0, 11, (3, 3))

         print(random_array_seed)
```

```
[[ 6  3 10]
 [ 7  4  6]
 [ 9  2  6]]
```

11.3 NumPy-Operations

11.3.1 List-like-Operations

Now that we know how to initialise an array, we want to work with it. Arrays can basically do everything a normal list can do, and more. Some common functions for lists are adding, inserting and deleting elements. For arrays, this is done using the following commands.

```
In [53]: numpy_array = np.arange(1, 11)  # 1D array from 1 to 10
         print(len(numpy_array), numpy_array)
```

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

Append an element at the end of an array.

```
In [54]: numpy_array = np.append(numpy_array, 11)
         print(len(numpy_array), numpy_array)
```

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

Insert an element at the second to last in the NumPy array

```
In [55]: numpy_array = np.insert(numpy_array, -1, 0)
         print(len(numpy_array), numpy_array)
```

```
12 [ 1  2  3  4  5  6  7  8  9 10  0 11]
```

Delete the second to last element of the NumPy array

```
In [56]: numpy_array = np.delete(numpy_array, -2)
         print(len(numpy_array), numpy_array)
```

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

To copy an array, we can't just use the = operator. That way we only copy the reference to the array, which means that if we change something in the copy, the source array will also be changed. To avoid this we use the `copy()` function. For nested arrays we also need to use `deepcopy()` as explained in the previous lecture.

```
In [57]: array = np.array([1, 2, 3, 4, 5])
         print(f"Array: {array}")

         # Copy a numpy array the WRONG way
         copy_of_array = array
         copy_of_array[0] = 100
         print(f"Copy of Array: {copy_of_array}")
         print(f"Array: {array}")

         # Copy a numpy array the RIGHT way
         copy_of_array = array.copy()
         copy_of_array[0] = 200
         print(f"Copy of Array: {copy_of_array}")
         print(f"Array: {array}")
```

```
Array: [1 2 3 4 5]
Copy of Array: [100   2   3   4   5]
Array: [100   2   3   4   5]
Copy of Array: [200   2   3   4   5]
Array: [100   2   3   4   5]
```

11.3.2 Mathematical Operations

We can use mathematical operations that we can apply to matrices with arrays. For example, multiplying two arrays of the same length. With lists, we would have to iterate through each element of both lists and multiply them. With an array, we can just multiply the whole array by another array, no slow loops needed.

```
In [58]: list1 = [1, 2, 3, 4, 5]
         list2 = [6, 7, 8, 9, 10]
         list3 = []

         for i in range(len(list1)):
             list3.append(list1[i] * list2[i])

         array1 = np.array(list1)
         array2 = np.array(list2)

         array3 = array1 * array2

         print(f"List 3: {list3}")
         print(f"Array 3: {array3}")
```

```
List 3: [6, 14, 24, 36, 50]
Array 3: [ 6 14 24 36 50]
```

With an array, you can use many mathematical functions without iterating through each element. Firstly we want to square each element of an array.

```
In [59]: # Other NumPy operations
         array = np.array(range(1, 6))
         print(f"Array: {array}")

         array_squared = np.square(array)
         print(f"Array squared: {array_squared}")
```

```
Array: [1 2 3 4 5]
Array squared: [ 1  4  9 16 25]
```

Next we want the root of every element.

```
In [60]: # Calculate the square root of every element
         print(f"Square root of Array: {np.sqrt(array_squared)}")
```

```
Square root of Array: [1. 2. 3. 4. 5.]
```

Lastly we want value of `sin(x)` for every element.

```
In [61]: print(f"Sin(Array): {np.sin(array)}")
```

```
Sin(Array): [ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427]
```

11.3.3 NumPy-Specific-Operations

When we have a two-dimensional array, it is sometimes necessary to access only one row or column of the whole matrix. This can be done by using `:` instead of a specific integer value as the index. We can also get slices of a row or a column by using `:`.

```
In [62]: # Initialise a 2D numpy array with 7 rows and 3 columns with values from 1 to 21
         two_d_array = np.arange(1, 22).reshape(7, 3)
         print(two_d_array)

         print(f"First Row: {two_d_array[0, :]}")

         print(f"Second Column: {two_d_array[:, 1]}")

         print(f"Part of Second Column: {two_d_array[1:5, 1]}")
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]
 [16 17 18]
 [19 20 21]]
First Row: [1 2 3]
Second Column: [ 2  5  8 11 14 17 20]
Part of Second Column: [ 5  8 11 14]
```

To mimic an if-else statement in an array, we can use the `np.where()` function. Instead of checking each element by iterating through the array, we can use the function to vectorise the process. The function takes the condition to be checked as input and returns the modified array. Additionally, we can specify as input what to do with the element depending on whether the condition is met or not.

```
In [63]: arr = np.array([10, 20, 30, 40, 50])

         result = np.where(arr > 30, 1, 0)
         print(result)
```

```
[0 0 0 1 1]
```

We can also use a `mask` for the condition. A mask is usually a boolean array that specifies where a condition is true or false.

```
In [64]: arr = np.array([10, 20, 30, 40, 50])

         mask = arr > 30

         # Apply the mask
         result = np.where(mask, 1, 0)
         print(result)
```

```
[0 0 0 1 1]
```

For a more complex mask, we want to filter out the diagonal of a matrix and set all values on it to 0. All other values should remain the same as before.

```
In [65]: matrix = np.ones((5, 5))
         print(matrix)

         idx1, idx2 = np.meshgrid(np.arange(0, 5), np.arange(0, 5), indexing="ij")
         mask = idx1 == idx2

         print(np.where(mask, 0, matrix))
```

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

11.4 NumPy in Heuristics

The Travelling Salesman Problem describes a salesman who has to travel from city to city to sell products and return home at the end of the day. The goal is to find the shortest path to visit each city exactly once. To solve this problem, we could use a model and solve it to optimality. But because solving it takes time, especially for large instances with many cities to visit, we want to solve it heuristically.

A little refresher. A heuristic is an algorithm that generates solutions to problems that may not be optimal, but are sufficient for the task. They should be much faster than the exact methods and are used to speed up the decision making process.

In the following we will implement a simple greedy heuristic to solve the TSP quickly. The idea of a greedy heuristic in this context is to choose the next city to travel to based on distance. The city closest to the current location is chosen as the next city to visit. In this way, we compute the complete tour by starting at our home, repeatedly adding the city closest to our current location, and adding the trip to our home when there are no more cities to visit. Below is the pseudocode we want to implement. Pseudocode is used to outline the logic of an algorithm using plain language and basic programming style syntax. It's meant to be easy to read and understand.

Algorithm 1 SimpleGreedyHeuristic(*instance*)

Initialise $sequence = [i0]$ and $ofv = 0$
Initialise $unvisited = [i1, i2, ..., iN]$
while $len(unvisited) \neq 0$ **do**
    Initialise $shortest = \infty$ and $next = None$
    for $j \in unvisited$ **do**
        if $distance[sequence[-1], j] \leq shortest$ **then**
            $shortest = distance[sequence[-1], j]$
            $next = j$
    Add $next$ to $sequence$
    Delete $next$ from $unvisited$
    $ofv = ofv + distance[sequence[-1], j]$
Add $i0$ to $sequence$
$ofv = ofv + distance[sequence[-1], i0]$
return $sequence, ofv$

To show how much faster NumPy can be when used efficiently, we implement the heuristic once with lists and once with arrays.

```
In [66]: import sys
         import math

         # adding data-folder with instance_module_tsp to the system path
         sys.path.append("../data")

         import instance_module_tsp
```

11.4.1 Greedy Heuristic for TSP with Lists

```
In [67]: def greedy_python(instance) -> tuple[list, float]:
             """
```

```
            Greedy algorithm to solve the shortest path problem. Travel to the nearest, not yet visited location and return to the first.

            Args:
                instance (Instance): An object of the class Instance with coordinates of the locations

            Returns:
                The sequence of locations and the total travel distance.
            """
            # Initialise the sequence with the start point
            sequence = [instance.locations[0]]
            travel_distance = 0
            unvisited = instance.locations[1:].copy()

            # While there are still locations to visit loop through the list of locations to find the shortest path from the most recent location
            while len(unvisited) != 0:
                # Initialise the shortest path and the found location
                shortest_path = math.inf
                found_j = "Nothing"

                # Loop through the list of locations to find the shortest path
                for j in unvisited:
                    if instance.c[sequence[-1], j] <= shortest_path:
                        shortest_path = instance.c[sequence[-1], j]
                        found_j = j

                # Append the found location to the sequence
                sequence.append(found_j)

                # Update the most recent location and the travel distance
                travel_distance += shortest_path

                # Remove the location from the list of not yet visited locations
                unvisited.remove(found_j)

            # Add the End Point to the sequence manually
            travel_distance += instance.c[sequence[-1], instance.locations[0]]
            sequence.append(instance.locations[0])

            return sequence, travel_distance
```

11.4.2 Greedy Heuristic for TSP with Arrays

```
In [68]: def greedy_numpy(instance, distance_matrix) -> tuple[list, float]:
            """
            Optimized Greedy algorithm to solve the shortest path problem. Travels to the nearest, not yet visited location
            and returns to the first. Utilizes NumPy for faster computation.

            Args:
```

```
            instance: An object of the class Instance with coordinates of the␣
↪locations

        Returns:
            The sequence of locations and the total travel distance.
        """
        # Initialise the sequence with the start point (index 0)
        sequence = [instance.locations[0]]
        most_recent = 0
        travel_distance = 0.0

        # Create an array of boolean flags to track visited locations
        to_visit = np.ones(instance.nb_locations + 1, dtype=bool)
        to_visit[0] = False  # Start point is already visited

        # Main loop to visit each location
        for _ in range(1, instance.nb_locations + 1):
            # Get the distances from the most recent location to all other locations
            distances_from_current = distance_matrix[
                most_recent, :
            ]  # Get distances from current location to all others

            # Mask to filter only unvisited locations
            distances_to_unvisited = np.where(to_visit, distances_from_current, np.
↪inf)

            # Find the nearest unvisited location
            next_location = np.argmin(distances_to_unvisited)
            shortest_path = distances_to_unvisited[next_location]

            # Update the sequence, travel distance, and visited status
            sequence.append(instance.locations[next_location])
            travel_distance += shortest_path
            most_recent = next_location
            to_visit[next_location] = False  # Mark the location as visited

        # Return to the start point (index 0)
        travel_distance += instance.c[
            instance.locations[most_recent], instance.locations[0]
        ]
        sequence.append(instance.locations[0])

        return sequence, travel_distance
```

11.4.3 Comparison of the two Implementations

```
In [69]: # Run the heuristics with a timer to compare the runtime
         def run_heuristics(
             instance, distance_matrix: np.array, iterations: int, all_heuristics: list
         ) -> None:
             """
             Function for running all heuristics given and time the runtime of each␣
↪heuristic for a given number of iterations.
```

```
        Args:
            instance (_type_): instance of the TSP problem
            distance_matrix (np.array): distance matrix of the TSP problem
            iterations (int): number of iterations to run the heuristics
            all_heuristics (list): list of all heuristics to run
        """

        # Define the steps to print the progress
        steps = [
            i - 1
            for i in range(int(iterations / 4), iterations, max(int(iterations /␣
↪4), 1))
        ]

        # Iterate over all heuristics
        for heuristic in all_heuristics:
            total_time = 0

            # Print the progress
            for i in range(iterations):
                if i in steps:
                    print(f"{i+1} iterations are done for {heuristic}")

                # Start the Timer
                start = perf_counter()

                # Call Algorithm
                if heuristic == "Greedy_Python":
                    sequence, ofv = greedy_python(instance)
                elif heuristic == "Greedy_NumPy":
                    sequence, ofv = greedy_numpy(instance, distance_matrix)

                # Stop the Timer
                end = perf_counter()

                total_time += end - start

            # Print the Time used for the Heuristic
            print(f"Time used for {heuristic}: {total_time} seconds")
            print(sequence)
            print(ofv)
```

```
In [70]: nb_locations = 300

        # Pick Instance
        instance = instance_module_tsp.Instance(nb_locations)

        # Initialise the Distance Matrix
        distance_matrix = np.ones((instance.nb_locations + 1, instance.nb_locations +␣
↪1))
        for index_i, i in enumerate(instance.locations):
            for index_j, j in enumerate(instance.locations):
                distance_matrix[index_i, index_j] = instance.c[i, j]
```

```
        iterations = 1000
        # ["Greedy_Python", "Greedy_NumPy"]
        all_heuristics = ["Greedy_Python", "Greedy_NumPy"]

        # Call and time the Heuristics (and the Model)
        run_heuristics(instance, distance_matrix, iterations, all_heuristics)
```

```
250 iterations are done for Greedy_Python
500 iterations are done for Greedy_Python
750 iterations are done for Greedy_Python
Time used for Greedy_Python: 11.863420250989293 seconds
['i0', 'i149', 'i203', 'i172', 'i195', 'i36', 'i272', 'i147', 'i150', 'i38', 'i57',␣
 ↪'i204', 'i220', 'i62', 'i58', 'i157', 'i196', 'i248', 'i208', 'i231', 'i291', 'i232',␣
 ↪'i175', 'i43', 'i130', 'i102', 'i21', 'i225', 'i281', 'i265', 'i84', 'i199', 'i27',␣
 ↪'i166', 'i122', 'i257', 'i28', 'i35', 'i271', 'i185', 'i117', 'i258', 'i34', 'i299',␣
 ↪'i181', 'i156', 'i200', 'i262', 'i276', 'i111', 'i244', 'i239', 'i255', 'i253', 'i7',␣
 ↪'i46', 'i270', 'i137', 'i112', 'i18', 'i256', 'i104', 'i100', 'i223', 'i63', 'i245',␣
 ↪'i132', 'i182', 'i229', 'i133', 'i50', 'i47', 'i105', 'i273', 'i86', 'i179', 'i11',␣
 ↪'i173', 'i49', 'i292', 'i151', 'i124', 'i227', 'i94', 'i120', 'i29', 'i74', 'i209',␣
 ↪'i186', 'i24', 'i69', 'i45', 'i4', 'i44', 'i70', 'i140', 'i143', 'i108', 'i290',␣
 ↪'i89', 'i162', 'i155', 'i23', 'i119', 'i226', 'i126', 'i37', 'i85', 'i218', 'i77',␣
 ↪'i30', 'i165', 'i153', 'i68', 'i31', 'i280', 'i148', 'i230', 'i295', 'i193', 'i88',␣
 ↪'i243', 'i55', 'i78', 'i287', 'i9', 'i60', 'i284', 'i158', 'i80', 'i14', 'i214',␣
 ↪'i259', 'i249', 'i212', 'i22', 'i171', 'i26', 'i159', 'i297', 'i71', 'i187', 'i59',␣
 ↪'i217', 'i129', 'i293', 'i246', 'i13', 'i250', 'i286', 'i65', 'i296', 'i109', 'i92',␣
 ↪'i10', 'i228', 'i275', 'i177', 'i282', 'i145', 'i279', 'i81', 'i233', 'i298', 'i294',␣
 ↪'i174', 'i106', 'i237', 'i32', 'i134', 'i118', 'i98', 'i170', 'i283', 'i39', 'i17',␣
 ↪'i138', 'i91', 'i285', 'i123', 'i99', 'i278', 'i206', 'i167', 'i183', 'i254', 'i194',␣
 ↪'i8', 'i25', 'i251', 'i154', 'i51', 'i269', 'i178', 'i191', 'i219', 'i15', 'i96',␣
 ↪'i107', 'i54', 'i210', 'i213', 'i95', 'i161', 'i115', 'i234', 'i20', 'i56', 'i221',␣
 ↪'i139', 'i266', 'i113', 'i131', 'i222', 'i2', 'i125', 'i116', 'i205', 'i201', 'i114',␣
 ↪'i169', 'i103', 'i188', 'i3', 'i242', 'i264', 'i224', 'i247', 'i93', 'i180', 'i236',␣
 ↪'i127', 'i79', 'i90', 'i289', 'i211', 'i97', 'i73', 'i215', 'i235', 'i128', 'i184',␣
 ↪'i75', 'i238', 'i136', 'i202', 'i160', 'i300', 'i163', 'i142', 'i197', 'i176', 'i207',␣
 ↪'i61', 'i135', 'i121', 'i198', 'i263', 'i189', 'i52', 'i42', 'i67', 'i144', 'i82',␣
 ↪'i66', 'i41', 'i164', 'i241', 'i274', 'i5', 'i146', 'i48', 'i33', 'i53', 'i64',␣
 ↪'i288', 'i19', 'i76', 'i240', 'i216', 'i40', 'i72', 'i260', 'i1', 'i192', 'i110',␣
 ↪'i12', 'i16', 'i6', 'i277', 'i101', 'i190', 'i267', 'i168', 'i152', 'i83', 'i252',␣
 ↪'i87', 'i261', 'i268', 'i141', 'i0']
839.5975941611277
250 iterations are done for Greedy_NumPy
500 iterations are done for Greedy_NumPy
750 iterations are done for Greedy_NumPy
Time used for Greedy_NumPy: 1.557720586004507 seconds
['i0', 'i149', 'i203', 'i172', 'i195', 'i36', 'i272', 'i147', 'i150', 'i38', 'i57',␣
 ↪'i204', 'i220', 'i62', 'i58', 'i157', 'i196', 'i248', 'i208', 'i231', 'i291', 'i232',␣
 ↪'i175', 'i43', 'i130', 'i102', 'i21', 'i225', 'i281', 'i265', 'i84', 'i199', 'i27',␣
 ↪'i166', 'i122', 'i257', 'i28', 'i35', 'i271', 'i185', 'i117', 'i258', 'i34', 'i299',␣
 ↪'i181', 'i156', 'i200', 'i262', 'i276', 'i111', 'i244', 'i239', 'i255', 'i253', 'i7',␣
 ↪'i46', 'i270', 'i137', 'i112', 'i18', 'i256', 'i104', 'i100', 'i223', 'i63', 'i245',␣
 ↪'i132', 'i182', 'i229', 'i133', 'i50', 'i47', 'i105', 'i273', 'i86', 'i179', 'i11',␣
 ↪'i173', 'i49', 'i292', 'i151', 'i124', 'i227', 'i94', 'i120', 'i29', 'i74', 'i209',␣
 ↪'i186', 'i24', 'i69', 'i45', 'i4', 'i44', 'i70', 'i140', 'i143', 'i108', 'i290',␣
 ↪'i89', 'i162', 'i155', 'i23', 'i119', 'i226', 'i126', 'i37', 'i85', 'i218', 'i77',␣
 ↪'i30', 'i165', 'i153', 'i68', 'i31', 'i280', 'i148', 'i230', 'i295', 'i193', 'i88',␣
 ↪'i243', 'i55', 'i78', 'i287', 'i9', 'i60', 'i284', 'i158', 'i80', 'i14', 'i214',␣
 ↪'i259', 'i249', 'i212', 'i22', 'i171', 'i26', 'i159', 'i297', 'i71', 'i187', 'i59',␣
 ↪'i217', 'i129', 'i293', 'i246', 'i13', 'i250', 'i286', 'i65', 'i296', 'i109', 'i92',␣
 ↪'i10', 'i228', 'i275', 'i177', 'i282', 'i145', 'i279', 'i81', 'i233', 'i298', 'i294',␣
 ↪'i174', 'i106', 'i237', 'i32', 'i134', 'i118', 'i98', 'i170', 'i283', 'i39', 'i17',␣
 ↪'i138', 'i91', 'i285', 'i123', 'i99', 'i278', 'i206', 'i167', 'i183', 'i254', 'i194',␣
 ↪'i8', 'i25', 'i251', 'i154', 'i51', 'i269', 'i178', 'i191', 'i219', 'i15', 'i96',␣
 ↪'i107', 'i54', 'i210', 'i213', 'i95', 'i161', 'i115', 'i234', 'i20', 'i56', 'i221',␣
 ↪'i139', 'i266', 'i113', 'i131', 'i222', 'i2', 'i125', 'i116', 'i205', 'i201', 'i114',␣
 ↪'i169', 'i103', 'i188', 'i3', 'i242', 'i264', 'i224', 'i247', 'i93', 'i180', 'i236',␣
 ↪'i127', 'i79', 'i90', 'i289', 'i211', 'i97', 'i73', 'i215', 'i235', 'i128', 'i184',␣
 ↪'i75', 'i238', 'i136', 'i202', 'i160', 'i300', 'i163', 'i142', 'i197', 'i176', 'i207',␣
 ↪'i61', 'i135', 'i121', 'i198', 'i263', 'i189', 'i52', 'i42', 'i67', 'i144', 'i82',␣
 ↪'i66', 'i41', 'i164', 'i241', 'i274', 'i5', 'i146', 'i48', 'i33', 'i53', 'i64',␣
 ↪'i288', 'i19', 'i76', 'i240', 'i216', 'i40', 'i72', 'i260', 'i1', 'i192', 'i110',␣
 ↪'i12', 'i16', 'i6', 'i277', 'i101', 'i190', 'i267', 'i168', 'i152', 'i83', 'i252',␣
 ↪'i87', 'i261', 'i268', 'i141', 'i0']
```

```
839.5975941611277
```

Chapter 12

NetworkX

"NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks." (Source: NetworkX)

```
In [68]: import networkx as nx
```

12.1 Types of Graphs

Networkx Class	Type	Self-loops allowed	Parallel edges allowed
Graph	undirected	Yes	No
DiGraph	directed	Yes	No
MultiGraph	undirected	Yes	Yes
MultiDiGraph	directed	Yes	Yes

Source: https://networkx.org/documentation/stable/reference/classes/index.html

12.2 Nodes

12.2.1 Add nodes to Graph

Create a simple graph `G` with 5 nodes without edges.

```
In [69]: G = nx.Graph()
```

Add node `g1` to the graph `G`.

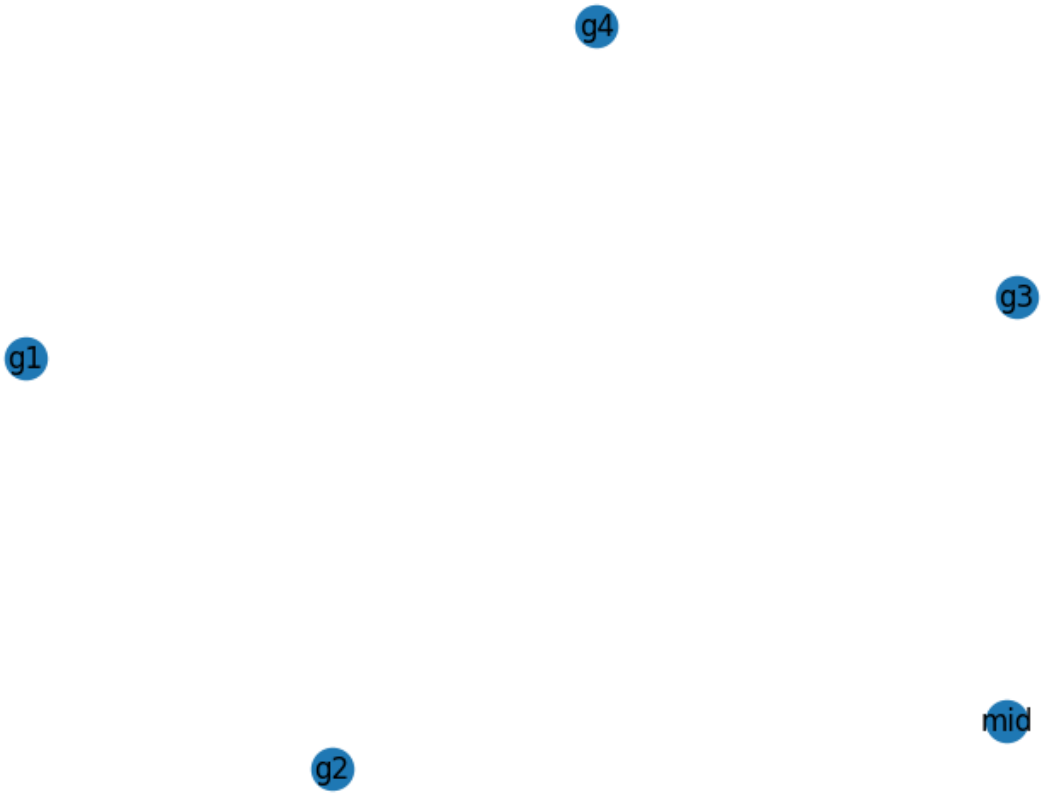
```
In [70]: G.add_node("g1")
```

To add more nodes than one at a time, use a list with all nodes you want to add to the graph `G`.

```
In [71]: my_nodes = ["g2", "g3", "g4", "mid"]
         G.add_nodes_from(my_nodes)
```

Draw the Graph `G` using `draw()`.

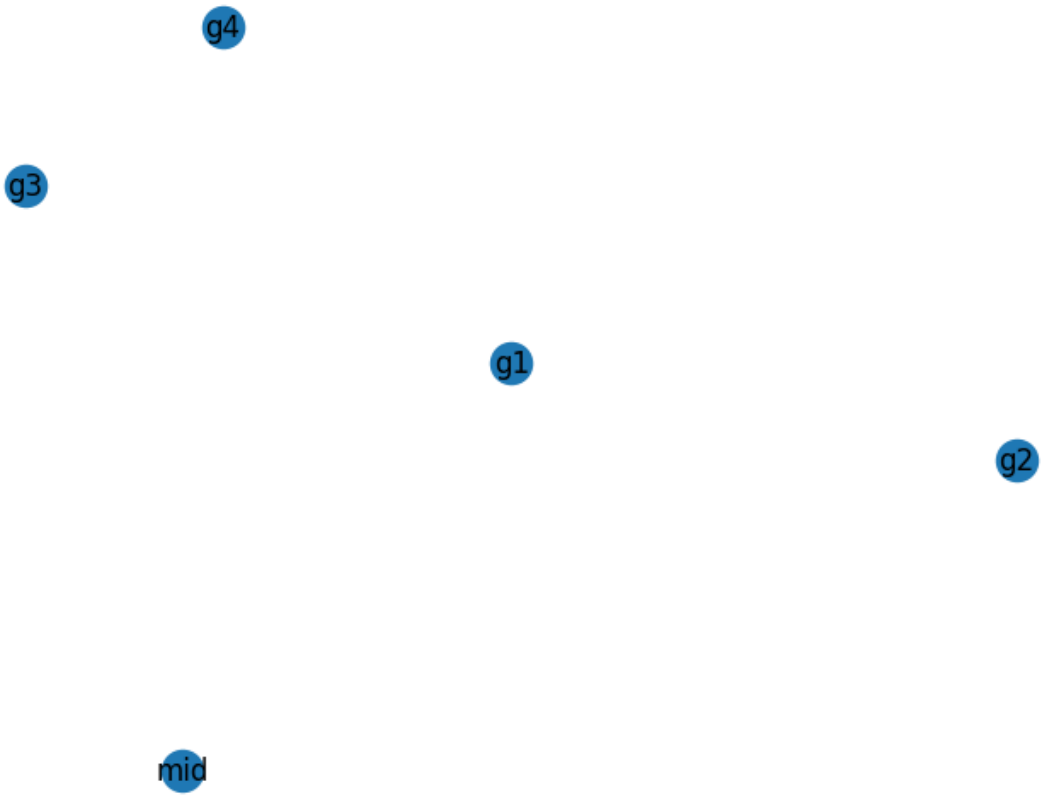
```
In [72]: nx.draw(G, with_labels=True)
```



12.2.2 Set Layout of Graph

There are predefined layouts for drawing graphs in matplotlib, e.g. `draw_circular`, `draw_shell` or `draw_random` (see documentation). The default layout is the spring layout. Some layouts (e.g. spring) will change if plotted again. Other layouts (e.g. circular) will always result in the same layout. To showcase we use a random layout for our graph.

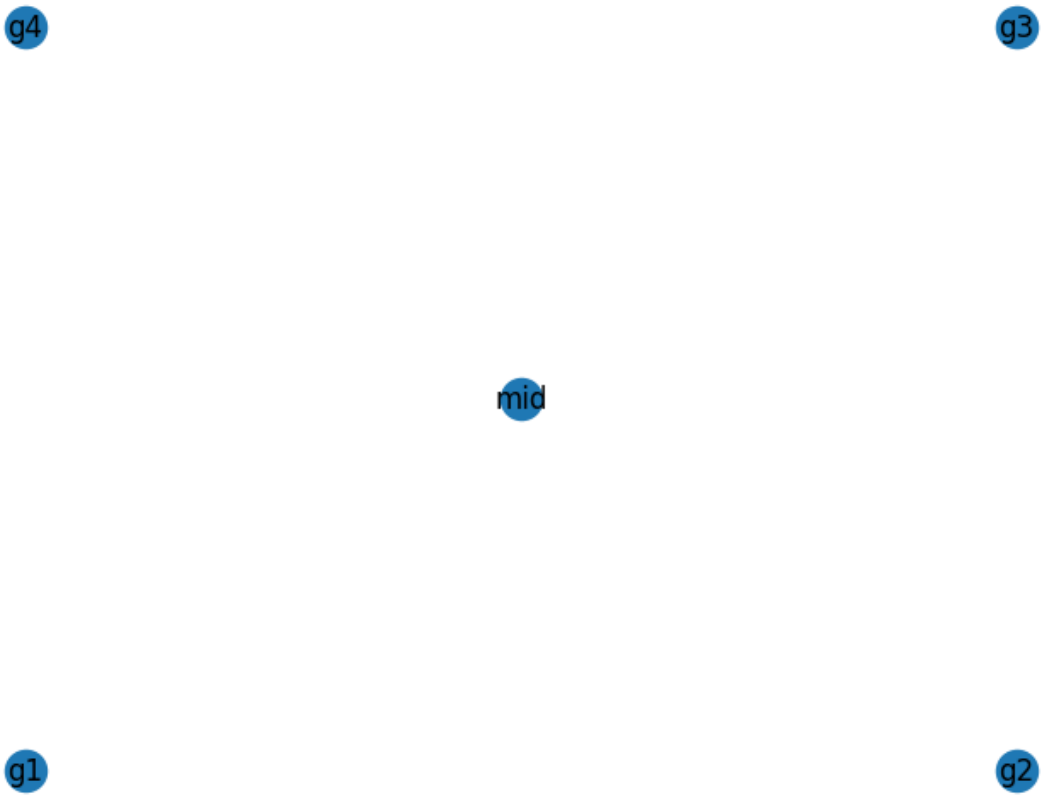
```
In [73]: nx.draw_random(G, with_labels=True)
```



12.2.3 Set Position of Nodes

In many cases, we have or want defined positions of the nodes. For these cases, we can add custom positions to the nodes. For this we use a dictionary with the node names as keys and the x,y-coordinates as a tuple.

```
In [74]: pos = {"g1": (0, 0), "g2": (4, 0), "g3": (4, 4), "g4": (0, 4), "mid": (2, 2)}

         nx.draw(G, pos, with_labels=True)
```



Let's assume we have a second graph `H` to which we want to add nodes from a dict. Within the dict, we have the positions of the nodes given. We can add the node position as an attribute to each node.

```
In [75]: my_nodes = {"g6": (2, 1), "g7": (2, 3)}

         H = nx.Graph()  # define a new graph

         for node, (x, y) in my_nodes.items():
             H.add_node(node, pos=(x, y))
```

If we don't have a dictionary with all nodes and positions, we can extract one from our graph, in which we added the position of a node as an attribute.

```
In [76]: pos_H = nx.get_node_attributes(H, "pos")
         print(pos_H)
         nx.draw(H, pos=pos_H, with_labels=True)
```

```
{'g6': (2, 1), 'g7': (2, 3)}
```

We can also set the position of the nodes in G afterwards with `set_node_attributes()`. Beforehand we just used the positions when drawing the graph. They were not added as attributes to each node.

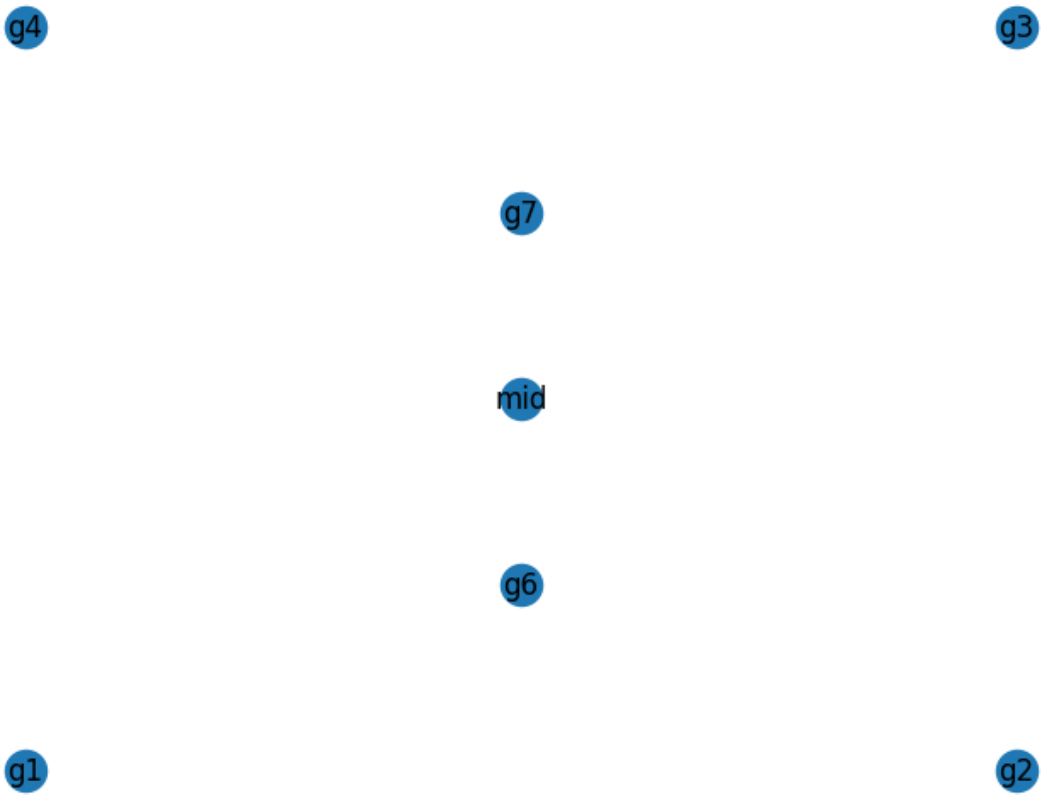
```
In [77]: nx.set_node_attributes(G, pos, name="pos")
```

Let's add the nodes from `H` to the graph `G`.

```
In [78]: G.add_nodes_from(H)
         nx.set_node_attributes(G, pos_H, name="pos")
```

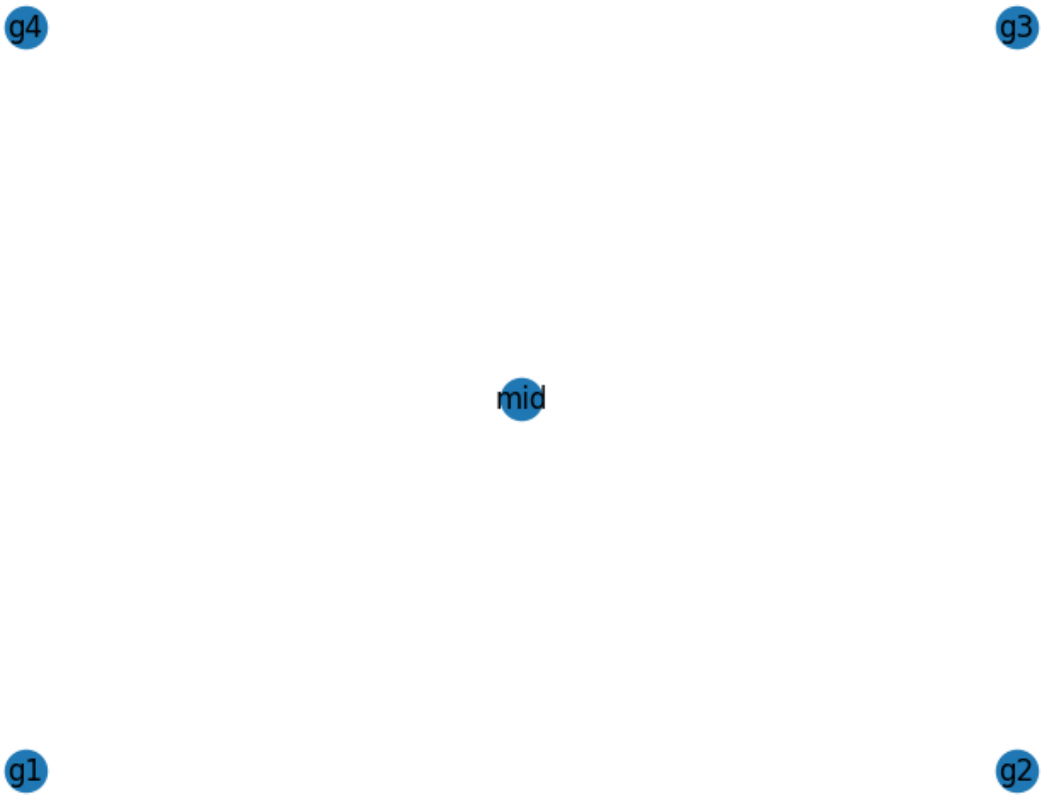
Print the graph G with the defined positions. We get all positions from the nodes in the graph G and use this dictionary as an argument in the `draw`-function

```
In [79]: pos_total = nx.get_node_attributes(G, "pos")
         nx.draw(G, pos=pos_total, with_labels=True)
```



We can also delete specific nodes from the graph (in a similar way to adding them).

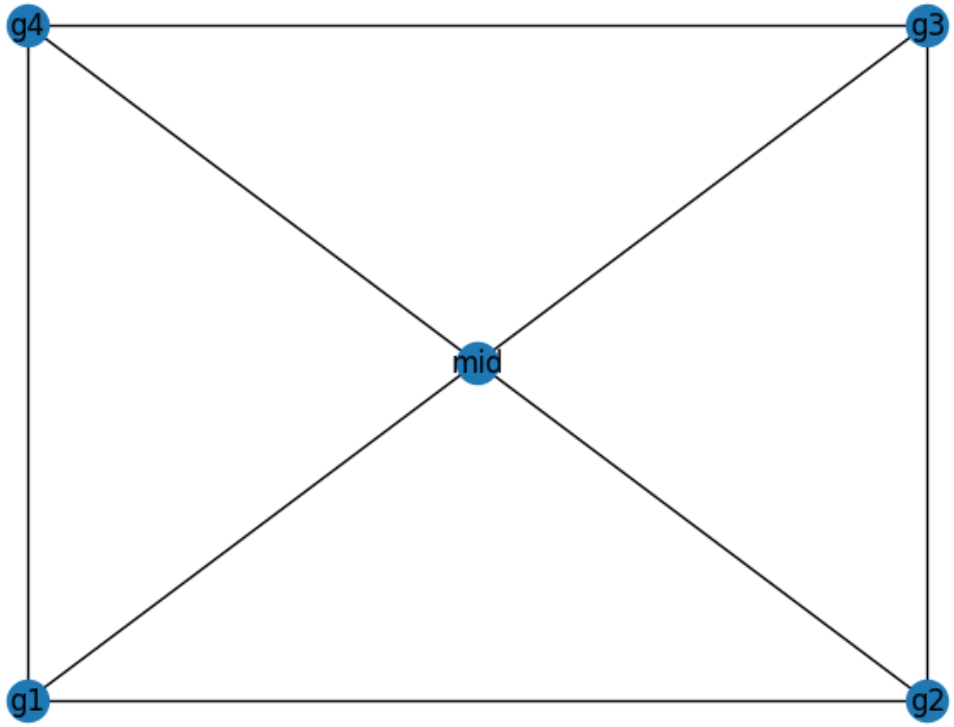
```
In [80]: G.remove_nodes_from(["g6", "g7"])
         nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True)
```



12.3 Edges

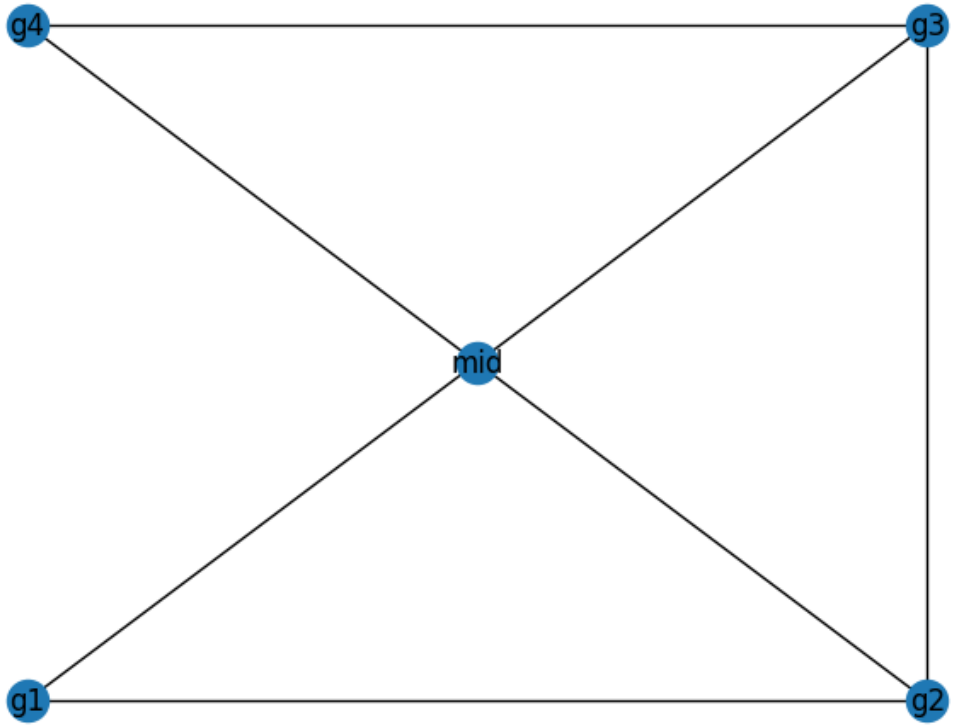
In addition to nodes, we can add edges to the graph. In this case, all edges are given in a list `my_edges`. Edges are defined by the two nodes which they connect. We can add the edges with `add_edges_from()`.

```
In [81]: my_edges = [
             ("g1", "g2"),
             ("g2", "g3"),
             ("g3", "g4"),
             ("g4", "g1"),
             ("g1", "mid"),
             ("g2", "mid"),
             ("g3", "mid"),
             ("g4", "mid"),
         ]

         G.add_edges_from(my_edges)

         nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True)
```



Of course, we can also delete edges.

```
In [82]: G.remove_edge("g1", "g4")
         nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True)
```



12.4 Inspecting the Graph

We can check the number of edges and nodes with `number_of_nodes()` and `number_of_edges()`. Or we can just print our Graph.

```
In [83]: nb_nodes = G.number_of_nodes()
         nb_edges = G.number_of_edges()

         print(f"Our graph has {nb_nodes} nodes and {nb_edges} edges.")

         print(G)
```

```
Our graph has 5 nodes and 7 edges.
Graph with 5 nodes and 7 edges
```

We can also have all nodes/edges displayed and the neighboring nodes.

```
In [84]: print(G.nodes)  # all nodes
         print((G.edges))  # all edges
         print(list(G.neighbors("g2")))  # all neighbors of node g2
         print(G.degree["g2"])  # degree of node g1 (number of edges incident on it)
```

```
['g1', 'g2', 'g3', 'g4', 'mid']
[('g1', 'g2'), ('g1', 'mid'), ('g2', 'g3'), ('g2', 'mid'), ('g3', 'g4'), ('g3', 'mid'),␣
↪('g4', 'mid')]
['g1', 'g3', 'mid']
3
```

12.5 Attributes

12.5.1 Get Attributes of Nodes

We can also print the nodes with its attributes.

```
In [85]: print(G.nodes)  # all nodes
         print(G.nodes(data=True))  # all nodes with attributes
```

```
['g1', 'g2', 'g3', 'g4', 'mid']
[('g1', {'pos': (0, 0)}), ('g2', {'pos': (4, 0)}), ('g3', {'pos': (4, 4)}), ('g4',␣
↪{'pos': (0, 4)}), ('mid', {'pos': (2, 2)})]
```

12.5.2 Add Color

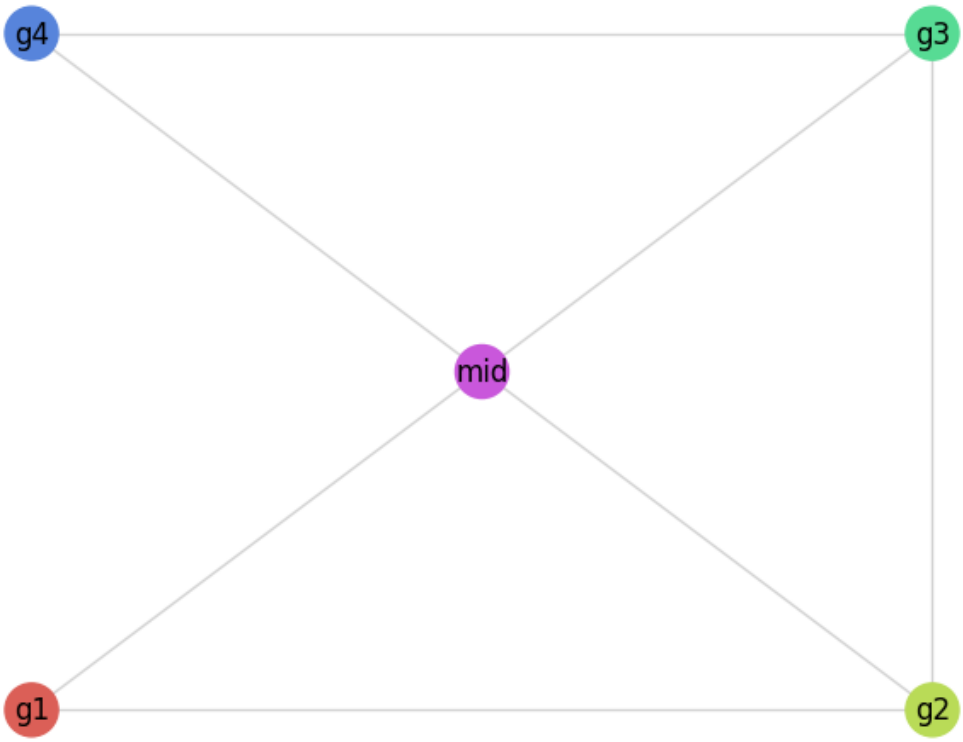
We can add additional attributes to our graph, e.g. a color for each node. We can use `sns.color_palette()` if we want to give each node a different color. We first create a color list with as many colors as nodes. Afterwards we set the color of each node with `G.nodes[node]["color"]`.

```
In [86]: import seaborn as sns

         color_list = sns.color_palette("hls", nb_nodes)

         for color, node in zip(color_list, G.nodes()):
             G.nodes[node]["color"] = color
```

Draw the graph using the defined positions `pos`, the attribute `color`, a node size of `500` and `lightgrey` as edge color.

Note that `node_color` takes a color or an array, not a dictionary.

```
In [87]: options = {
             "node_size": 500,
             "node_color": list(nx.get_node_attributes(G, "color").values()),
             "edge_color": "lightgrey",
         }

         nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True, **options)
```



12.6 DiGraphs and working with Graphs

12.6.1 Initialize Graph from CSV-File

We have a csv dataset of edges of a graph given. The edges are directed and have a different weights. Between some nodes, we have edges in both directions while for other nodes, there exists only an edge in one direction.

We can open the csv-file and parse it to our Graph `G`. As a delimiter we used `,` in our csv-files. We want to create a directed Graph, so we use `nx.DiGraph`. Our nodes are named with strings and we have two different attributes given for each edge with it's weight and edge color.

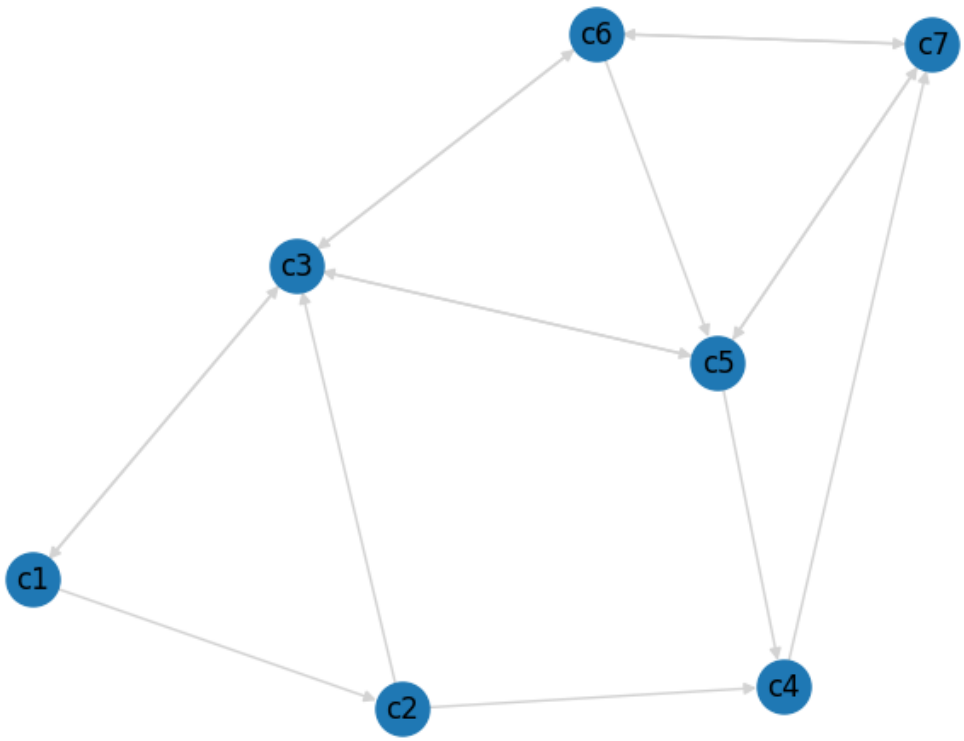
```
In [88]: edges = open("../data/edge_list.csv", "r")
         next(edges, None)  # skip the first line in the input file

         G = nx.parse_edgelist(
             edges,
             delimiter=",",
             create_using=nx.DiGraph,
             nodetype=str,
             data=(
                 ("Distance", float),
                 ("EdgeColor", str),
             ),
```

```
        )

        print(G)
        print(list(G.edges(data=True)))
```

```
DiGraph with 7 nodes and 16 edges
[('c1', 'c2', {'Distance': 2.0, 'EdgeColor': 'lightgrey'}), ('c1', 'c3', {'Distance': 6.
↪0, 'EdgeColor': 'lightgrey'}), ('c2', 'c4', {'Distance': 2.0, 'EdgeColor':␣
↪'lightgrey'}), ('c2', 'c3', {'Distance': 3.0, 'EdgeColor': 'lightgrey'}), ('c4', 'c7',␣
↪{'Distance': 7.0, 'EdgeColor': 'lightgrey'}), ('c7', 'c6', {'Distance': 6.0,␣
↪'EdgeColor': 'lightgrey'}), ('c7', 'c5', {'Distance': 4.0, 'EdgeColor': 'lightgrey'}),␣
↪('c3', 'c6', {'Distance': 2.0, 'EdgeColor': 'lightgrey'}), ('c3', 'c5', {'Distance': 1.
↪0, 'EdgeColor': 'lightgrey'}), ('c3', 'c1', {'Distance': 2.0, 'EdgeColor':␣
↪'lightgrey'}), ('c6', 'c5', {'Distance': 3.0, 'EdgeColor': 'lightgrey'}), ('c6', 'c7',␣
↪{'Distance': 4.0, 'EdgeColor': 'lightgrey'}), ('c6', 'c3', {'Distance': 4.0,␣
↪'EdgeColor': 'lightgrey'}), ('c5', 'c4', {'Distance': 6.0, 'EdgeColor': 'lightgrey'}),␣
↪('c5', 'c7', {'Distance': 8.0, 'EdgeColor': 'lightgrey'}), ('c5', 'c3', {'Distance': 5.
↪0, 'EdgeColor': 'lightgrey'})]
```

After parsing the edge list we have all the information in our Graph `G` and can just draw it as usual.

```
In [89]: options = {
             "node_size": 500,
             "edge_color": list(nx.get_edge_attributes(G, "EdgeColor").values()),
         }

         nx.draw(G, with_labels=True, **options)
```



`Note:` If you just draw the graph without setting the edgecolor in the options, the edgecolor is set to default black. Even though you initialized the edgecolor as an attribute in the graph.

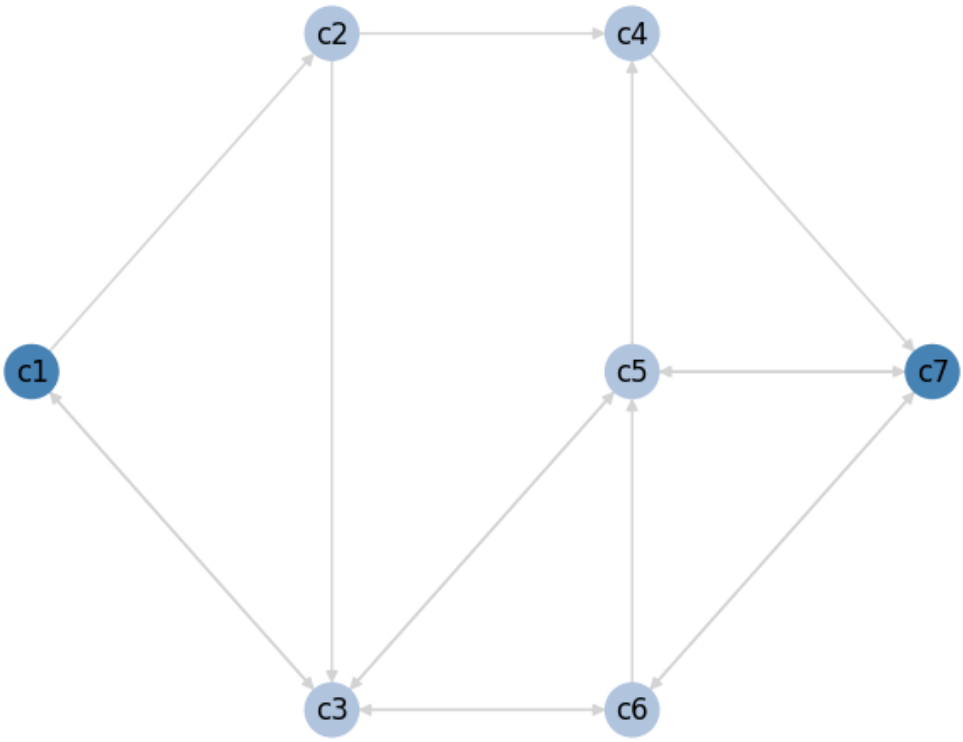
In addition, we have a dictionary with information about node attributes given. We can add this information to the nodes.

```
In [90]: attributes = {
             "c1": {"color": "steelblue", "pos": (0, 1)},
             "c2": {"color": "lightsteelblue", "pos": (1, 2)},
             "c3": {"color": "lightsteelblue", "pos": (1, 0)},
             "c4": {"color": "lightsteelblue", "pos": (2, 2)},
             "c5": {"color": "lightsteelblue", "pos": (2, 1)},
             "c6": {"color": "lightsteelblue", "pos": (2, 0)},
             "c7": {"color": "steelblue", "pos": (3, 1)},
         }

         nx.set_node_attributes(G, attributes)
         # as check: access the attribute color of the first node
         print(G.nodes["c1"]["color"])
```

```
steelblue
```

We can now draw the graph with the corresponding node and edge colors and the given positions of the node.

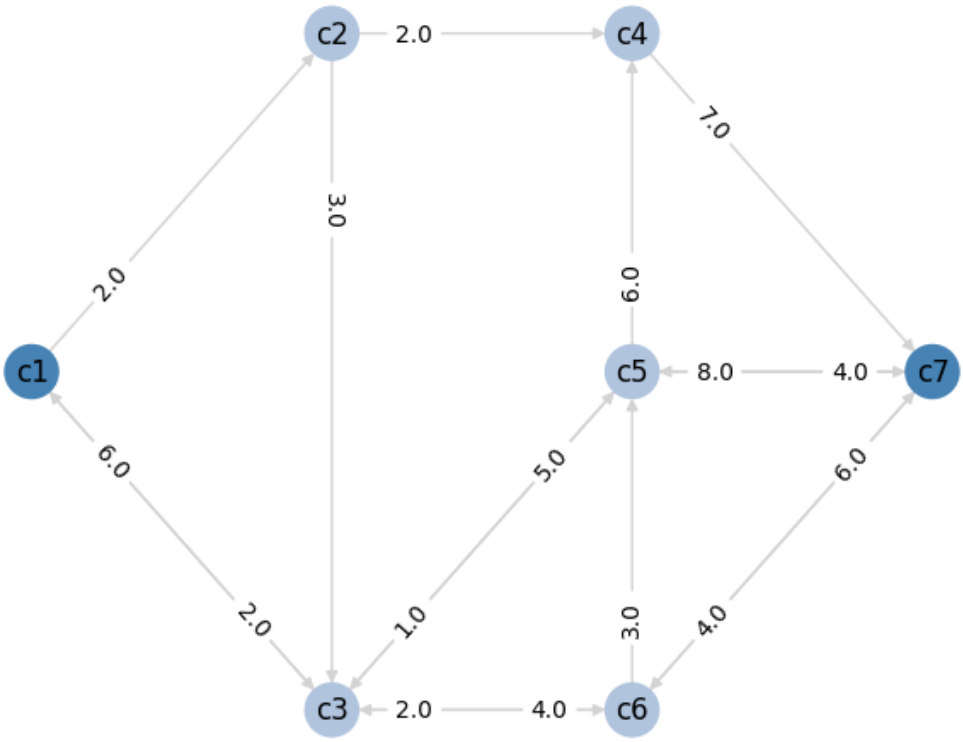
```
In [91]: options = {
             "node_color": list(nx.get_node_attributes(G, "color").values()),
             "node_size": 500,
             "edge_color": list(nx.get_edge_attributes(G, "EdgeColor").values()),
         }

         nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True, **options)
```



We also defined the distance as an attribute of an edge. To see those distances we use another `draw`-function on top. As a base we draw the graph as before and add the `edge_labels` afterwards with the function `draw_networkx_edge_labels`. To see the different labels for the directions we say the label should be positioned on 0.25 of length of the edge.

12.6.2 Add Edge Labels to Graph

```
In [92]: pos = nx.get_node_attributes(G, "pos")
         edge_labels = nx.get_edge_attributes(G, "Distance")

         nx.draw(G, pos, with_labels=True, **options)

         # _ is used to suppress the output which is shown in the terminal
         _ = nx.draw_networkx_edge_labels(G, pos, edge_labels, label_pos=0.25)
```



12.6.3 Shortest Path in Graph

Somebody would like to travel from the start node `c1` to the destination `c7`. Compute the shortest path and the length of this shortest path using `dijkstra_path()`.

More information about computing shortest paths: https://networkx.org/documentation/stable/reference/algorithms/sho (Make sure you check what kind of graph you have).

```
In [93]: path1 = nx.dijkstra_path(G, source="c1", target="c7", weight="Distance")
         print(f"The shortest path is {path1}.")
```

```
The shortest path is ['c1', 'c2', 'c4', 'c7'].
```

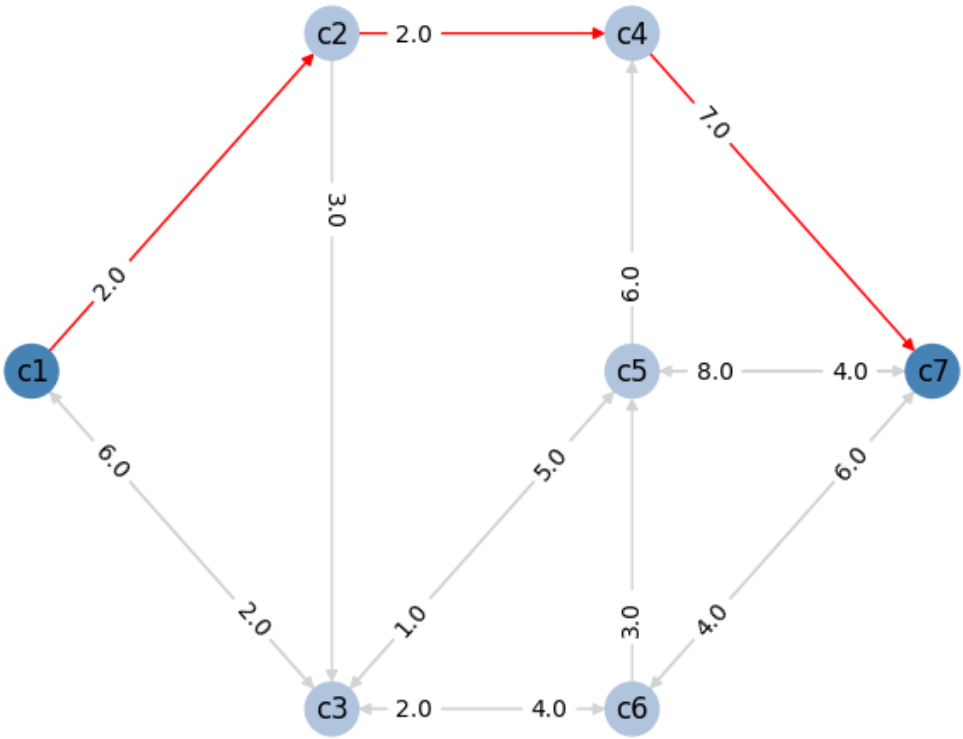
```
In [94]: length = nx.dijkstra_path_length(G, source="c1", target="c7", weight="Distance")
         print(f"The length of the shortest path is {length}.")
```

```
The length of the shortest path is 11.0.
```

Color the edges within the shortest path in red.

```
In [95]: for start_node, end_node in zip(path1[:-1], path1[1:]):
             G[start_node][end_node]["EdgeColor"] = "red"
```

```
        options["edge_color"] = list(nx.get_edge_attributes(G, "EdgeColor").values())

        nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True, **options)
        _ = nx.draw_networkx_edge_labels(
            G,
            pos=nx.get_node_attributes(G, "pos"),
            edge_labels=nx.get_edge_attributes(G, "Distance"),
            label_pos=0.25,
        )
```



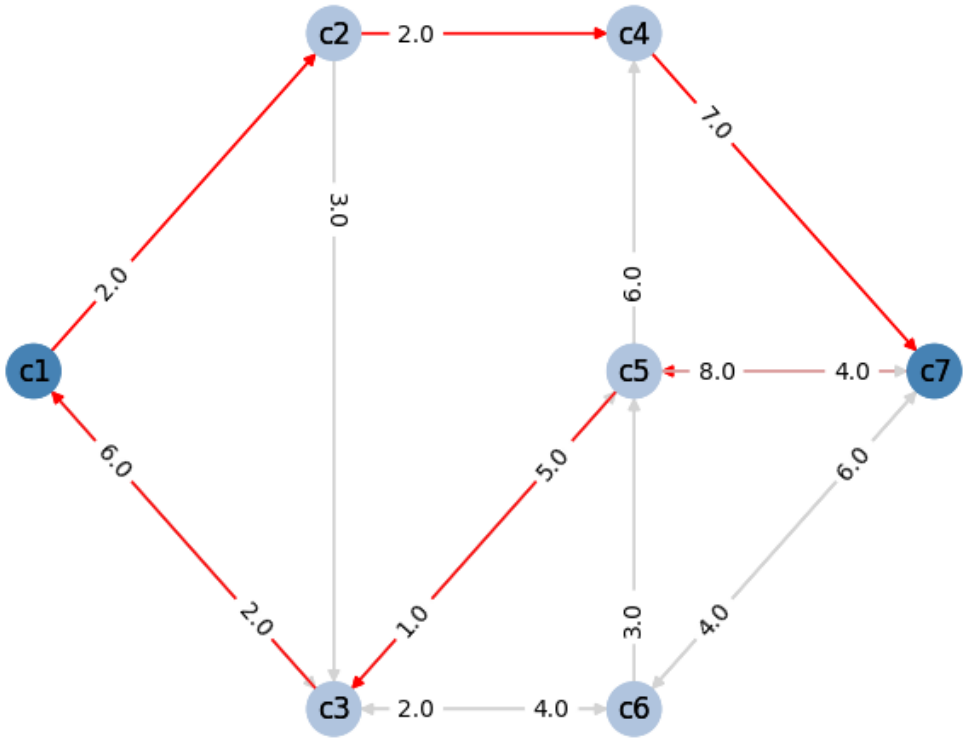
Compute the shortest round trip from node `c1` to `c7` and back to `c1`.

`Round Trip`: The shortest path from a starting point to an end point and back without visiting the same location twice.

```
In [96]: # you do not need to write source and target, however, it is less prone to␣
 ↪errors
        path2 = nx.dijkstra_path(G, "c7", "c1", weight="Distance")
        path_total = path1 + path2[1:]
        print(f"The shortest path is {path_total}.")
```

```
The shortest path is ['c1', 'c2', 'c4', 'c7', 'c5', 'c3', 'c1'].
```

```
In [97]: for start_node, end_node in zip(path_total, path_total[1:]):
             G[start_node][end_node]["EdgeColor"] = "red"

         options["edge_color"] = list(nx.get_edge_attributes(G, "EdgeColor").values())

         nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True, **options)
         nx.draw(G, pos=nx.get_node_attributes(G, "pos"), with_labels=True, **options)
         _ = nx.draw_networkx_edge_labels(
             G,
             pos=nx.get_node_attributes(G, "pos"),
             edge_labels=nx.get_edge_attributes(G, "Distance"),
             label_pos=0.25,
         )
```



Compute all simple paths within the graph from node `c1` to `c7`.

```
In [98]: for path in nx.all_simple_paths(G, source="c1", target="c7"):
             print(path)
```

```
['c1', 'c2', 'c4', 'c7']
['c1', 'c2', 'c3', 'c6', 'c5', 'c4', 'c7']
['c1', 'c2', 'c3', 'c6', 'c5', 'c7']
['c1', 'c2', 'c3', 'c6', 'c7']
```

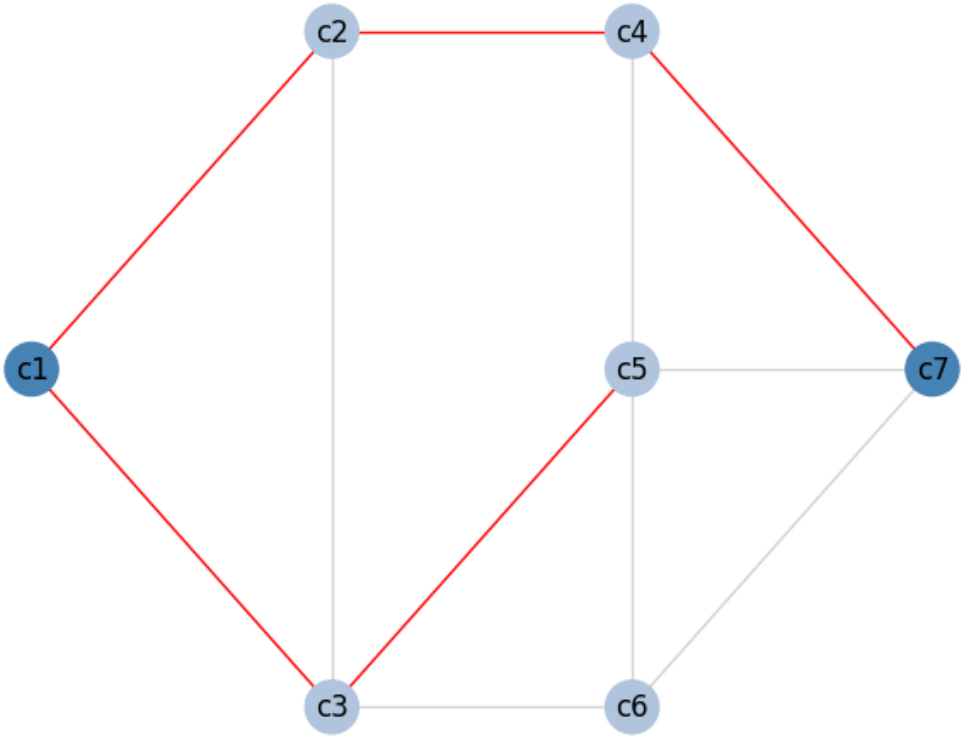
```
['c1', 'c2', 'c3', 'c5', 'c4', 'c7']
['c1', 'c2', 'c3', 'c5', 'c7']
['c1', 'c3', 'c6', 'c5', 'c4', 'c7']
['c1', 'c3', 'c6', 'c5', 'c7']
['c1', 'c3', 'c6', 'c7']
['c1', 'c3', 'c5', 'c4', 'c7']
['c1', 'c3', 'c5', 'c7']
```

12.6.4 Transform into undirected Graph

Transform the graph `G` into an undirected graph `H` using `to_undirected`. Using this function deletes parallel edges as you can see when printing the number of edges.

```
In [99]: H = G.to_undirected()
         options["edge_color"] = list(nx.get_edge_attributes(H, "EdgeColor").values())

         print(G)
         print(H)

         nx.draw(H, pos=nx.get_node_attributes(H, "pos"), with_labels=True, **options)
```

```
DiGraph with 7 nodes and 16 edges
Graph with 7 nodes and 11 edges
```



Get all neighbors of the node `c2` using `all_neighbors()`.

```
In [100]: neighbors = nx.all_neighbors(H, node="c2")
          for n in neighbors:
              print(n)
```

```
c1
c4
c3
```

There are many other functions and algorithms when working with graphs in `networkX`. Check the website: https://networkx.org/documentation/stable/reference/index.html

Chapter 13

Hill Climbing

13.1 General Things

The Hill Climbing algorithm is a simple heuristic algorithm for optimization problems. It is a local search algorithm that starts with an arbitrary solution to a problem and then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found. (Source: https://en.wikipedia.org/wiki/Hill_climbing) Also known as greedy local search, gradient search, first best neighbourhood-search and other names.

```
In [1]: import networkx as nx
        import copy
        import random
        from geopy.distance import geodesic as GD
```

13.2 TSP-Test-Instances

We define a test instance for the TSP with german cities and use their coordinates for the distances and visualization.

```
In [2]: test_instance = {
            "Munich": (48.137154, 11.576124),
            "Berlin": (52.518611, 13.408333),
            "Hamburg": (53.550556, 9.993333),
            "Frankfurt": (50.110556, 8.682222),
            "Stuttgart": (48.775556, 9.182222),
            "Dusseldorf": (51.233333, 6.783333),
            "Dortmund": (51.516667, 7.466667),
            "Bremen": (53.07516, 8.80777),
            "Dresden": (51.05089, 13.73832),
            "Leipzig": (51.33962, 12.37129),
            "Hannover": (52.37589, 9.73201),
            "Nuernberg": (49.44778, 11.06833),
            "Bielefeld": (52.03333, 8.53333),
            "Bonn": (50.73438, 7.09549),
            "Munster": (51.96236, 7.62571),
            "Karlsruhe": (49.00472, 8.38583),
            "Mannheim": (49.49671, 8.47955),
            "Augsburg": (48.37154, 10.89851),
```

```
        "Braunschweig": (52.26594, 10.52673),
        "Chemnitz": (50.83333, 12.91667),
        "Kiel": (54.32133, 10.13489),
        "Aachen": (50.77664, 6.08342),
        "Halle": (51.5, 12.0),
    }

    small_instance = {
        "Hamburg": (53.550556, 9.993333),
        "Munich": (48.137154, 11.576124),
        "Berlin": (52.518611, 13.408333),
        "Frankfurt": (50.110556, 8.682222),
        "Stuttgart": (48.775556, 9.182222),
    }
```

13.3 Solution-representation of the TSP

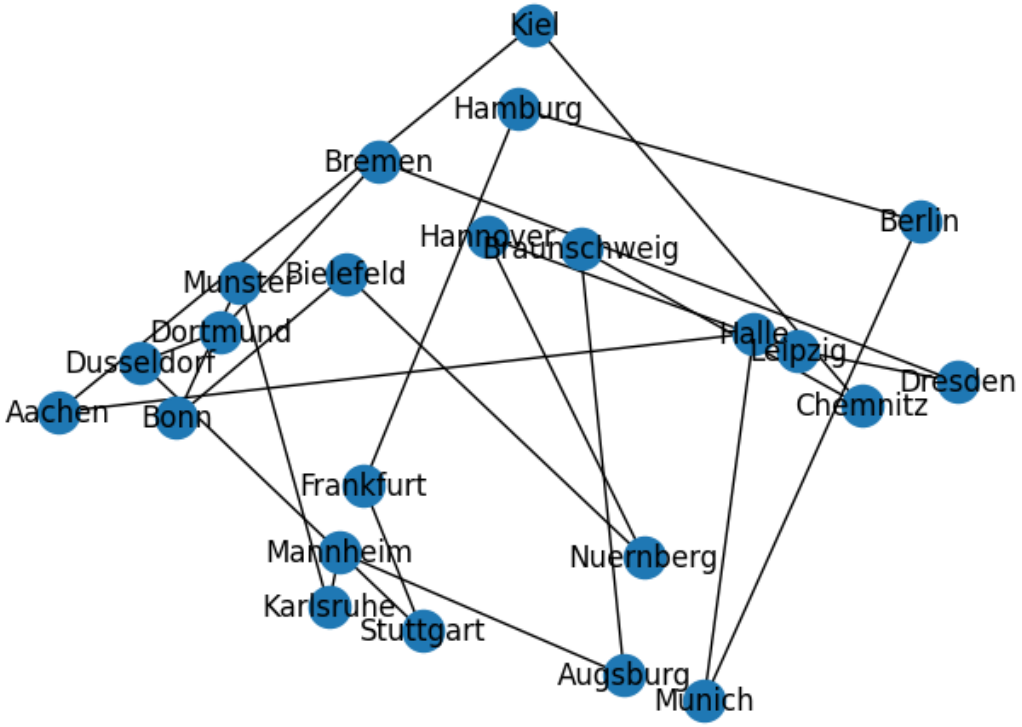
We define a class to create a solution for the TSP, using the Hill Climbing algorithm. With the `__init__`-function we create our object `TSP_solution`, which gets a Graph-Object and a path as input. We check whether the solution is feasible and create a new one if this is not the case. We check feasibility with a function of the class called `is_feasible()`, which covers all needed cases for proving feasibility. Furthermore we define two neighbourhood-operators with the functions `swap_cities` and `swap_edges`. To visualize the resulting graph we define the function `draw`.

```
In [ ]: class TSP_solution:
            def __init__(self, G: nx.Graph, path=None):
                self.G = G

                if not self.is_feasible():
                    # generate a feasible solution, start with a circle
                    if path is None:
                        path = list(
                            self.G.nodes
                        )  # visit all cities in the order, they are added to the graph
                    if len(path) == len(self.G.nodes):
                        path += [path[0]]  # close the circle

                    for city_origin, city_destination in zip(path[:-1], path[1:]):
                        distance = GD(
                            self.G.nodes[city_origin]["pos"],
                            self.G.nodes[city_destination]["pos"],
                        ).km

                        self.G.add_edge(city_origin, city_destination, Distance=distance)

                # the length of the path is the sum of the distances
                self.length = sum(nx.get_edge_attributes(self.G, "Distance").values())

            def swap_cities(self, cities: tuple):
                """
                Swap two cities in the path and change self.length

                Args:
```

```
                cities (tuple): two cities
            Returns:
                bool: True if the swap leads to a feasible solution
            """

            # Get the neighbouring cities of the cities to be swapped
            neighbors = {city: list(self.G.neighbors(city)) for city in cities}

            # Iterate over both sequences of the two cities
            for city, other_city in zip(cities, cities[::-1]):
                # Remove the edges between the cities and their neighbours unless␣
↪the neighbour is the other city
                for neighbour in neighbors[city]:
                    # Continue if the neighbour is the other city because we would␣
↪add the edge again later and undirected graph
                    if other_city == neighbour:
                        continue
                    self.length -= self.G[city][neighbour]["Distance"]
                    self.G.remove_edge(city, neighbour)

            # Do the same as above but add the edges between the other_city and the␣
↪neighbours of city
            for city, other_city in zip(cities, cities[::-1]):
                for neighbour in neighbors[city]:
                    if other_city == neighbour:
                        continue
                    self.G.add_edge(
                        other_city,
                        neighbour,
                        Distance=GD(
                            self.G.nodes[other_city]["pos"], self.G.
↪nodes[neighbour]["pos"]
                        ).km,
                    )

                    self.length += self.G[other_city][neighbour]["Distance"]
            return True

        def swap_edges(self, edges: tuple):
            """
            Swap two edges in the graph and change self.length

            Args:
                cities (tuple): two edges
            Returns:
                bool: True if the swap leads to a feasible solution
            """

            # Get the start and destination cities of the edges
            new_edges = [(orig, dest) for orig, dest in zip(*edges)]

            # Remove the edges and subtract the distance from the length
            for orig, dest in edges:
                self.length -= self.G[orig][dest]["Distance"]
```

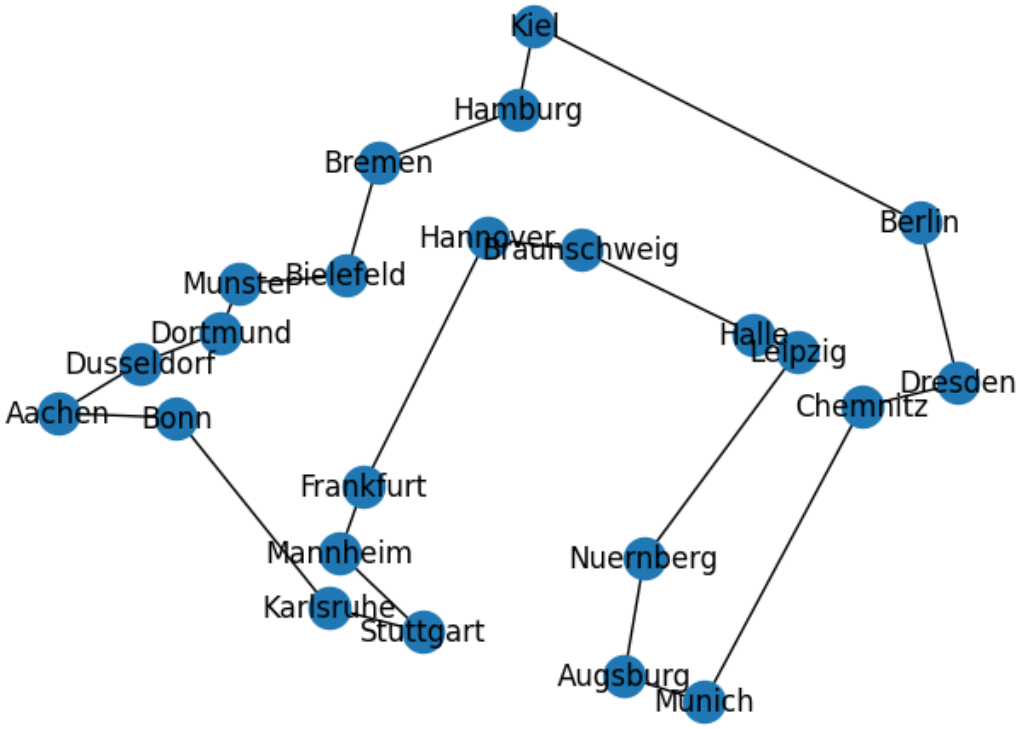
```
                    self.G.remove_edge(orig, dest)

                # Add the new edges and add the distance to the length
                for orig, dest in new_edges:
                    self.G.add_edge(
                        orig,
                        dest,
                        Distance=GD(self.G.nodes[orig]["pos"], self.G.
↪nodes[dest]["pos"]).km,
                    )
                    self.length += self.G[orig][dest]["Distance"]

                # Check if the solution is feasible
                return self.is_feasible()

            def draw(self):
                """
                Draw the graph to the solution with labels and positions
                """

                nx.draw(self.G, pos=nx.get_node_attributes(self.G, "pos"),␣
↪with_labels=True)

            def __repr__(self):
                return f"Path:{self.G.edges} \nwith: {self.length=}."

            def is_feasible(self):
                """
                Check if the solution is feasible. The solution is feasible if the graph
                is connected and the number of nodes is equal to the number of edges

                Returns:
                    bool: True if the solution is feasible
                """

                if not nx.is_connected(self.G):
                    return False

                if len(self.G.nodes) != len(self.G.edges):
                    return False

                return True
```

```
In [ ]: G = nx.Graph()  # define a new graph

        # Initialize the Graph with all cities and their positions as attributes
        for node, (y, x) in test_instance.items():
            G.add_node(node, pos=(x, y))

        # Create a first solution and draw it
        my_tsp = TSP_solution(G)
        my_tsp.draw()
```




```
In [ ]: # Start the optimization process
        for iteration in range(1, 10000):
            my_other_tsp = copy.deepcopy(my_tsp)
            is_feasible = True

            # Decide which operator to use based on a random number
            if random.random() < 0.3:
                # Choose two random cities to swap and swap them
                two_cities = random.sample(list(my_other_tsp.G.nodes), 2)
                is_feasible = my_other_tsp.swap_cities(two_cities)
            else:
                # Choose two random edges to swap and swap them
                two_edges = random.sample(list(my_other_tsp.G.edges), 2)
                is_feasible = my_other_tsp.swap_edges(two_edges)

            # If the solution is feasible and the new solution is better, update the␣
↪current best solution
            if is_feasible and my_other_tsp.length < my_tsp.length:
                my_tsp = my_other_tsp

        print(my_tsp)

        # Draw the final solution
        my_tsp.draw()
```

```
Path:[('Munich', 'Augsburg'), ('Munich', 'Chemnitz'), ('Berlin', 'Dresden'), ('Berlin',␣
 ↪'Kiel'), ('Hamburg', 'Kiel'), ('Hamburg', 'Bremen'), ('Frankfurt', 'Mannheim'),␣
 ↪('Frankfurt', 'Hannover'), ('Stuttgart', 'Karlsruhe'), ('Stuttgart', 'Mannheim'),␣
 ↪('Dusseldorf', 'Aachen'), ('Dusseldorf', 'Dortmund'), ('Dortmund', 'Munster'),␣
 ↪('Bremen', 'Bielefeld'), ('Dresden', 'Chemnitz'), ('Leipzig', 'Halle'), ('Leipzig',␣
 ↪'Nuernberg'), ('Hannover', 'Braunschweig'), ('Nuernberg', 'Augsburg'), ('Bielefeld',␣
 ↪'Munster'), ('Bonn', 'Karlsruhe'), ('Bonn', 'Aachen'), ('Braunschweig', 'Halle')]
with: self.length=3335.688719233649.
```



Instead of programming a heuristic for the TSP yourself, NetworkX contains a approximation function for this specific problem. You can find the documentation for this function here https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.approximation.traveling_ if you want to try it yourselves. Just keep in mind that even for this general problem our heuristic finds a better solution than the build-in-function. When the problem gets more specific the difference increases more and more, which is why it's important to not always use build-in-functions.

Chapter 14

Debugging in VS Code

Please clone the repository TSP_Heuristic from the following link: https://gitlab.uni-hannover.de/timohelfers/tsp_heuristic_students , as we work with this repository to learn about debugging.

14.1 Getting Started

14.1.1 What is Debugging?

Running your code results in one of three cases:

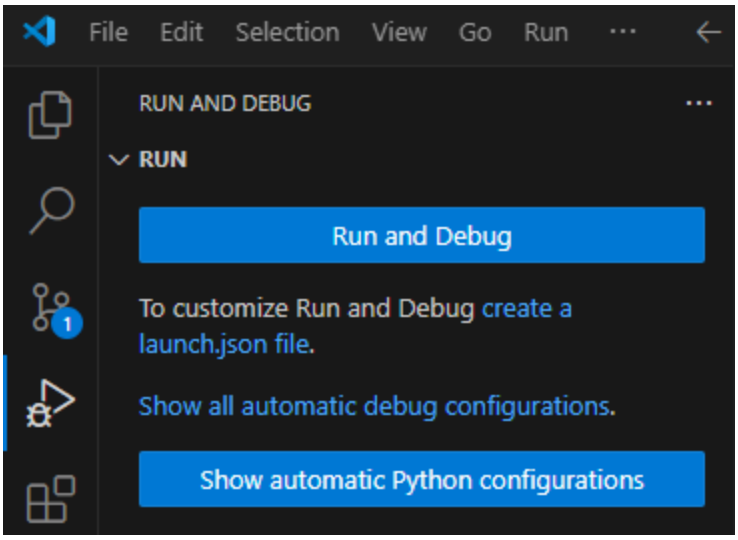
1. Code runs and generates the expected output
2. Code runs but behaves in an unintended way/generates unexpected output
3. Code returns an error

If the first case is true for you, congratulations. You should confirm this by testing different inputs but after that you can start applying your code for the intended purpose. If you are stuck in one of the other two cases, like we all are at some point, this lecture is for you.

Firstly it's completely normal to create errors and bugs during programming. Most importantly don't blame your PC. It just does what we told it to do. So look at your code from the perspective "where could I be wrong?". Sounds simple but sometimes debugging gets frustrating for example when you check a loop for the sixth time all though you are completely sure it should does what you expect and want it do.

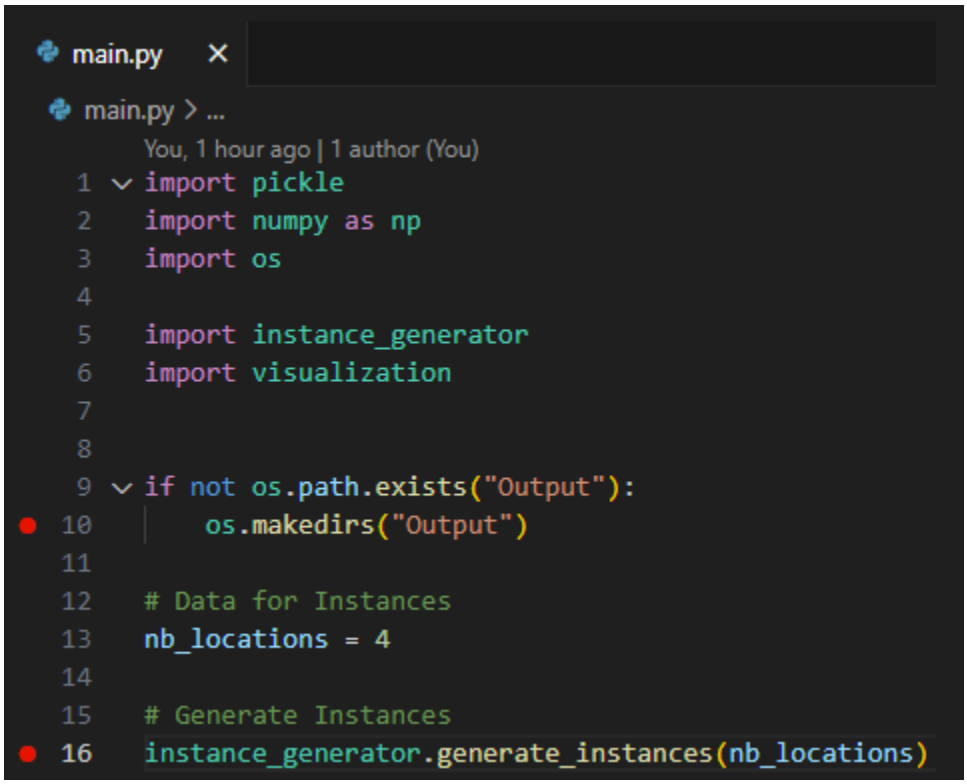
After this little intro, what do we really mean by debugging? Debugging can be defined as the process of identifying and fixing bugs or errors in your code. This means we try to narrow down the cause of a bug until we can identify it and change our code to fix it. Often times it's more or less clear in which function we need to look closely. For example if we have a heuristic for the TSP and the sequence of locations we get as output doesn't contain all locations in our instance, we know the heuristic function is at fold and not the function for visualisation. So we can narrow down our search on this function and the functions that are called in it.

But how can we check our variables and their changes during a run? Most of you did most certainly use `print()` statements of an variable and checked how it changes in the console during a run. Using `print()` statements is simple and can be very quick for small programs, but it has limitations. First of all your code and your output can become very messy very fast. Instead of getting a better overview it could get hard to read and understand. Especially if you are not sure what variable causes problems in the first place. The recommended method is to use Debugging tools. Those are often already integrated in your Code Editor. In VS Code this looks as follows:



14.1.2 Run and Debug

Before we click on `Run and Debug` we need to place breakpoints in our code-files. This is done by clicking left of the numbering of a line in the file. This creates a red dot to which the code is executed in the debugging tool. The code only stops before this line if this line is the next to be executed. If it gets skipped because of an `If`-Statement for example, the breakpoint is also skipped.

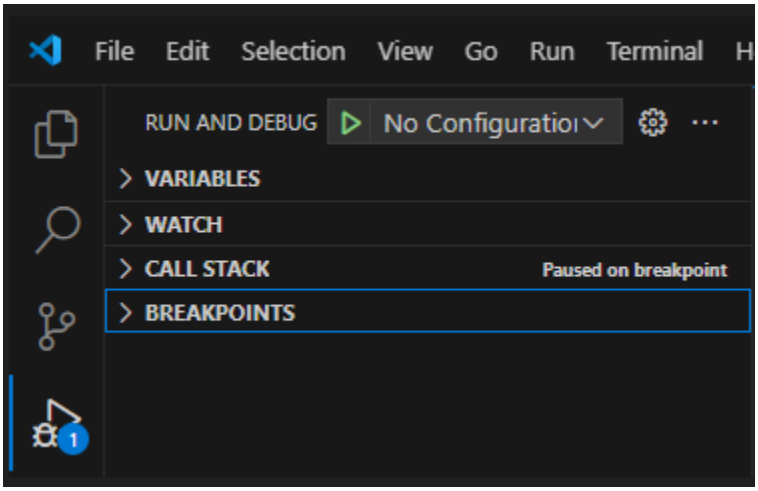


In this example when we click `Run and Debug` the code only stops before executing line 16. In the top we get a command bar. With this we can control how we want to go through the code.

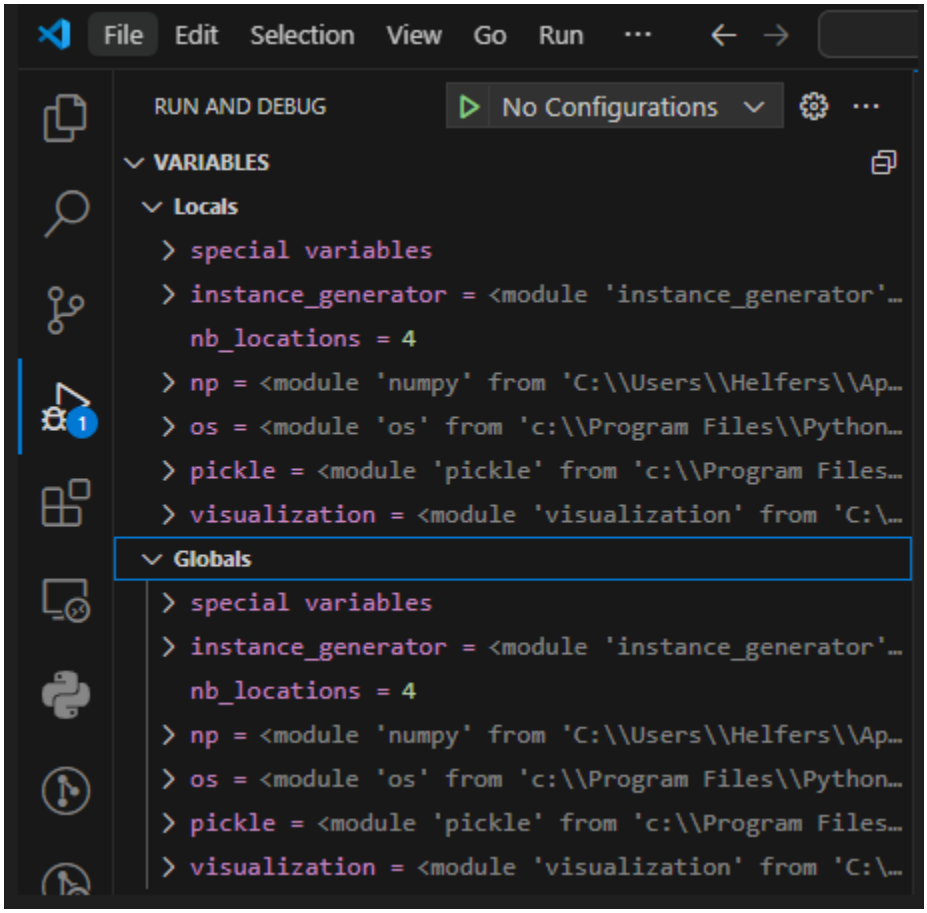
The six buttons work as follows:

1. `Continue`: Clicking will execute the code until the next breakpoint or until the code finishes.
2. `Step Over`: Clicking will execute the current line but without going into function calls. If there is a function call on the current line, the function will still be executed, but the debugger won't go into the function's internal details.
3. `Step Into`: Clicking will move the debugger into the first line of the called function or method. This allows you to inspect the internal workings of the function and see the input parameters, as well as how the function operates. It will only execute one line.
4. `Step Out`: Clicking will execute the remainder of the current function until it returns to its caller, pausing the debugger at the line immediately after the function call in the caller code.
5. `Restart`: Clicking will restart the debugging process from the beginning of the program. It stops the current session and then immediately starts a fresh debugging session.
6. `Stop`: Clicking will stop the debugging session entirely and terminate the program, halting all execution and exiting the debug mode.

Now that we hit our first breakpoint, we get four "folders".

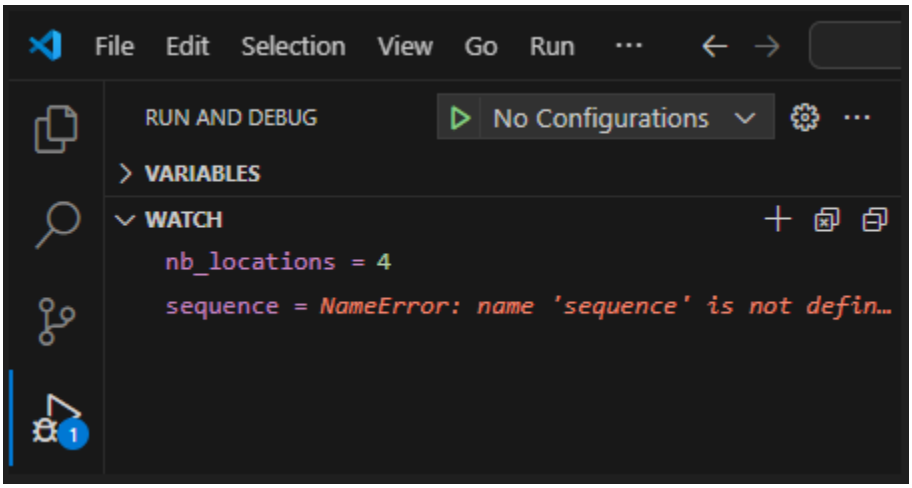


In `Variables` we can see all defined variables. By clicking on the folder we get `Locals` and `Globals`. Local variables are defined in a function or a loop and are only accessible inside the function or the block they are declared in. After stepping out of the function the debugger has no information on those variables. Global variables on the other hand are usually defined at the top level of a program and can be observed throughout the program, making them easy to track. As we are currently in the main-file all variables are local and global.

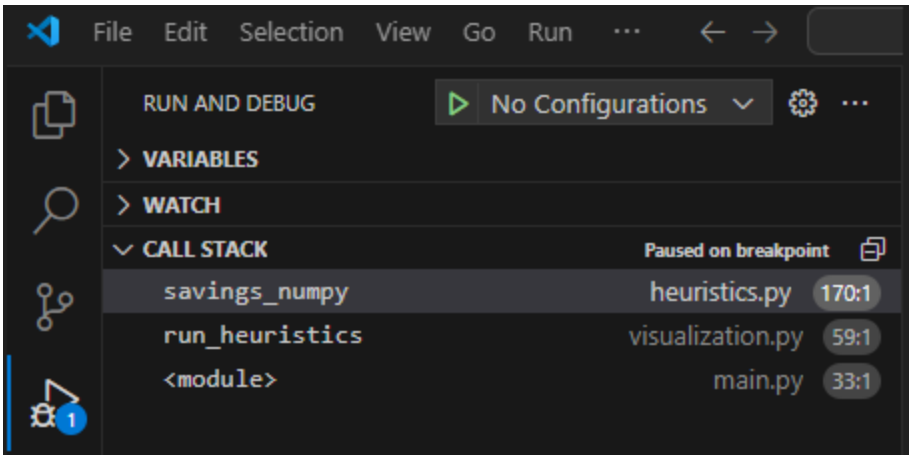


Every trackable variable is listed with there respective values, types and other specific informations.

If we want to watch a variable more closely we can use `Watch`. We can add variable names to this folder by clicking on the plus sign in it and getting the same informations as in `Variables`. We can also track variables which only exist in a function and don't have a value at all times. `Watch` helps keeping the focus on the variables you need to analyze.



`Call Stack` is the third folder and there we can see the path of how we got to this breakpoint in the code. If we add a breakpoint in the `savings_numpy` function and start debugging the `Call Stack` looks like followed:



This tells us we are in the function `savings_numpy` which is located in the file `heuristics.py` on line 170. This function is called by the function `run_heuristics` in `visualization.py` in line 59, which is again called by our main-file in line 33. Recreating the path from where a function is called sometimes helps identifying bugs but always increases understanding of your own code.

The last folder are the `Breakpoints`. This is self explanatory as it's all the breakpoints you added to all files in the folder. It helps manage your breakpoints as you can easily discard or toggle them instead of searching and deleting each one.

14.2 Procedure

You can look at debugging step by step.

1. Clear understanding of what the code is supposed to do:
    - Ensure you fully understand the intended functionality and logic behind the code. Knowing the expected behaviour is crucial for identifying bugs.
2. Inputs and outputs should be known/comprehensible for your test cases:
    - Define and understand the test cases, including the inputs and expected outputs. This helps you assess whether the program's behaviour is correct.
3. Set breakpoints near areas where you expect issues:
    - Based on error messages, logical analysis, or past experiences, place breakpoints in areas where you suspect issues. This allows you to pause and inspect the code at critical moments.
4. Start debugging: Step Into, Step Over, or Step Out of functions:
    - Use Step Into to dive into functions, Step Over to execute without going into function internals, and Step Out to exit functions. Adjust breakpoints as you go, and check variable values to verify the program's behaviour at each step.
5. Narrow down the issue:
    - As you gather information, focus on specific parts of the code where things go wrong. Consider potential causes such as incorrect input values, faulty conditions, or incorrect loop bounds.
6. Identify the problem:

- Pinpoint the exact issue causing the incorrect behaviour. This step involves understanding why the code is malfunctioning, whether due to logic, syntax, or other factors.

7. Fix the issue:

   - Once identified, implement a fix for the issue. Ensure your change logically resolves the problem and doesn't introduce new bugs.

8. Rerun and repeat if necessary:

   - Test the code again to ensure the problem is resolved. If the issue persists or new bugs arise, repeat the debugging process until the code works as expected.

14.3 Debug your first program

We deliberately added some bugs and errors in the branch "debugging" of the mentioned repository "TSP_Heuristic". The heuristics where already topic of our fifth lecture so you should be familiar with them. Try to find the bugs by using the debugging tool of VS Code and the information provided by the error messages in the terminal. To see what is expected from the code look into branch main and let it run with the same number of locations and iterations as in your debugging branch.

```
In [ ]: # 1. Start of sequence just 0 instead of "i0" causing error message KeyError:␣
 ↪(0, 'i1')
        # 2. found_j is not deleted in greedy_python causing an endless while-loop
        # 3. Range is wrong in for-loop of greedy_numpy causing "i2" to never be visited
        # 4. Savings get calculated wrong in savings_python causing the wrong tours to␣
 ↪be connected and therefore a worse OFV. Tricky to spot.
```

Chapter 15

Matplotlib and Pandas I

15.1 Simple Plots

First, we import `matplotlib`, its common to import directly the `pyplot` module as `plt`.

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
```

Plotting in `matplotlib` is very similar to how you would plot manually on paper. Simply give the function a series of x and y values and it will draw them and connect them with a lines.

Lets create a list of numbers from `-2` to `2` and a list of the corresponding square value.

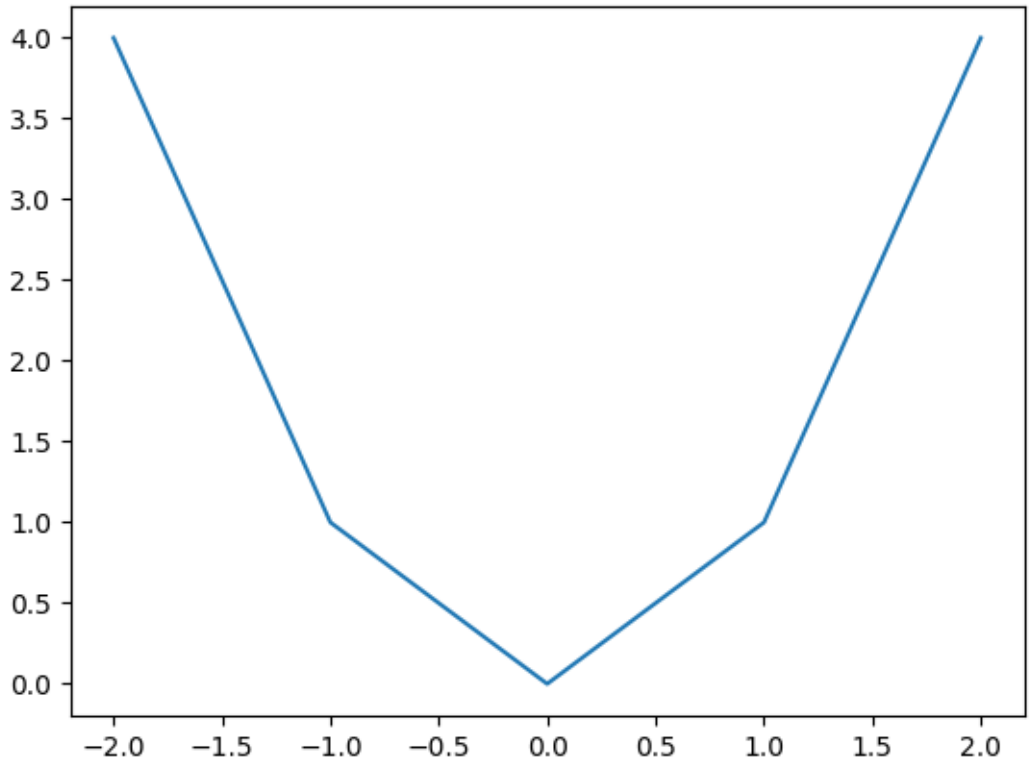
```
In [2]: x_values = list(range(-2, 3))

        # Compute the square of each number and store in a new list
        y_values = [num**2 for num in x_values]

        print("x:", x_values)
        print("y:", y_values)
```

```
x: [-2, -1, 0, 1, 2]
y: [4, 1, 0, 1, 4]
```

We can create a simple plot of this numbers using `plt.plot`. As we are using Jupiter notebook, we don*t need to use the `show()` function, but if you are using a different IDE you will need to use it.

```
In [3]: plt.plot(x_values, y_values)
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x7faf41986a90>]
```



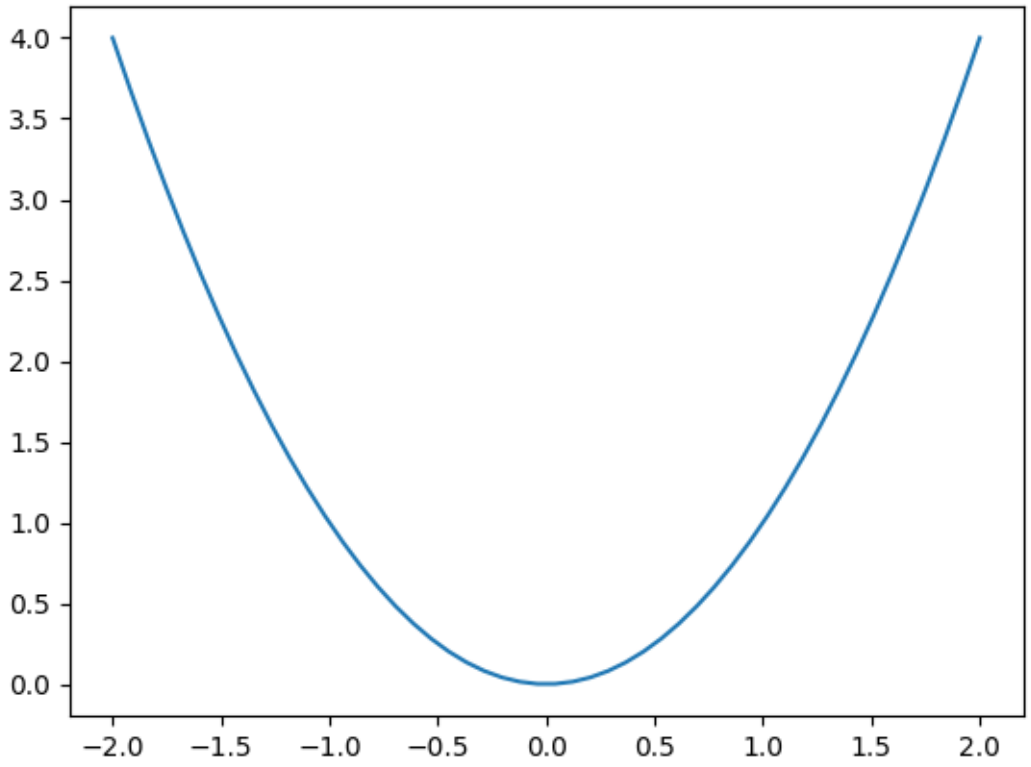
For a nice and even curves, we can use `np.linspace()` to generate a list of evenly spaced numbers.

```
In [4]: x_numpy = np.linspace(-2, 2)
        y_numpy = x_numpy**2

        print("x:", x_numpy)
        print("y:", y_numpy)

        plt.plot(x_numpy, y_numpy)
```

```
x: [-2.         -1.91836735 -1.83673469 -1.75510204 -1.67346939 -1.59183673
 -1.51020408 -1.42857143 -1.34693878 -1.26530612 -1.18367347 -1.10204082
 -1.02040816 -0.93877551 -0.85714286 -0.7755102  -0.69387755 -0.6122449
 -0.53061224 -0.44897959 -0.36734694 -0.28571429 -0.20408163 -0.12244898
 -0.04081633  0.04081633  0.12244898  0.20408163  0.28571429  0.36734694
  0.44897959  0.53061224  0.6122449   0.69387755  0.7755102   0.85714286
  0.93877551  1.02040816  1.10204082  1.18367347  1.26530612  1.34693878
  1.42857143  1.51020408  1.59183673  1.67346939  1.75510204  1.83673469
  1.91836735  2.        ]
y: [4.00000000e+00 3.68013328e+00 3.37359434e+00 3.08038317e+00
 2.80049979e+00 2.53394419e+00 2.28071637e+00 2.04081633e+00
 1.81424406e+00 1.60099958e+00 1.40108288e+00 1.21449396e+00
 1.04123282e+00 8.81299459e-01 7.34693878e-01 6.01416077e-01
 4.81466056e-01 3.74843815e-01 2.81549354e-01 2.01582674e-01
 1.34943773e-01 8.16326531e-02 4.16493128e-02 1.49937526e-02
 1.66597251e-03 1.66597251e-03 1.49937526e-02 4.16493128e-02
```

```
8.16326531e-02 1.34943773e-01 2.01582674e-01 2.81549354e-01
3.74843815e-01 4.81466056e-01 6.01416077e-01 7.34693878e-01
8.81299459e-01 1.04123282e+00 1.21449396e+00 1.40108288e+00
1.60099958e+00 1.81424406e+00 2.04081633e+00 2.28071637e+00
2.53394419e+00 2.80049979e+00 3.08038317e+00 3.37359434e+00
3.68013328e+00 4.00000000e+00]
```

```
Out[4]: [<matplotlib.lines.Line2D at 0x7faf41a5ee10>]
```

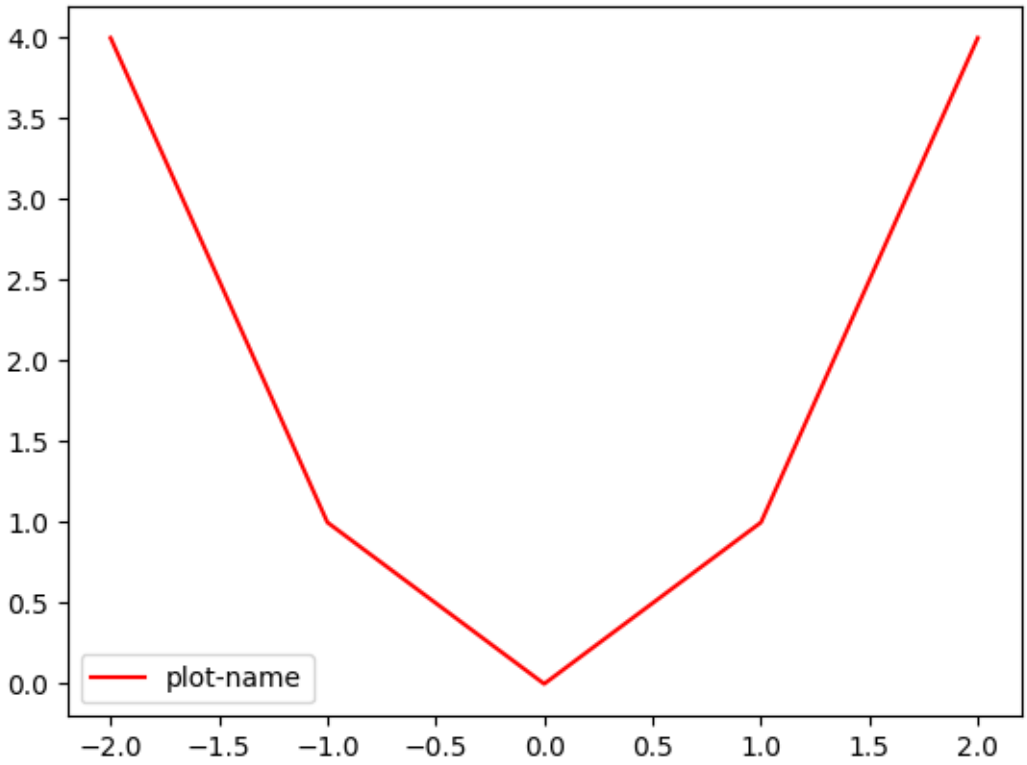


There are a lot of ways to customize the plot, here are a few examples. Check out the cheat sheet for a better overview.

Let's add a legend and change the color of the line to `red`.

```
In [5]: plt.plot(x_values, y_values, label="plot-name", color="red")
        plt.legend()
```

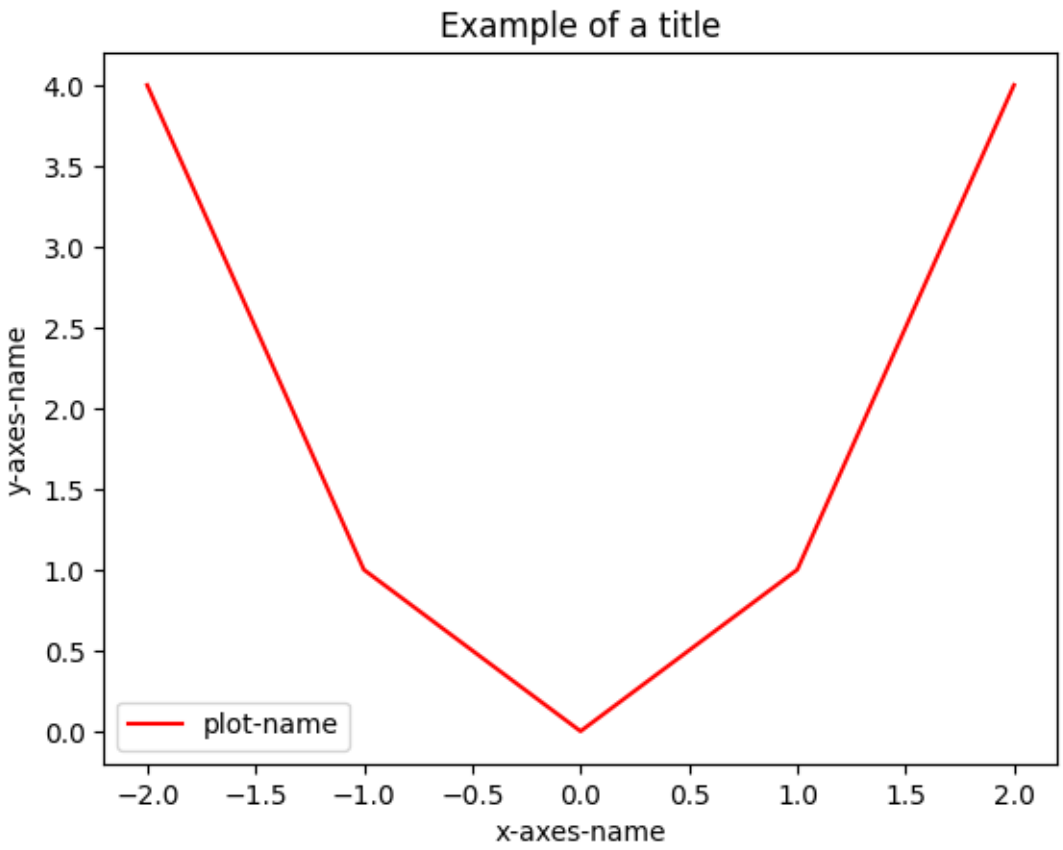
```
Out[5]: <matplotlib.legend.Legend at 0x7faf3f790e10>
```



We can also add a title and labels to the axes.

```
In [6]: plt.plot(x_values, y_values, label="plot-name", color="red")
        plt.legend()
        plt.xlabel("x-axes-name")
        plt.ylabel("y-axes-name")
        plt.title("Example of a title")
```

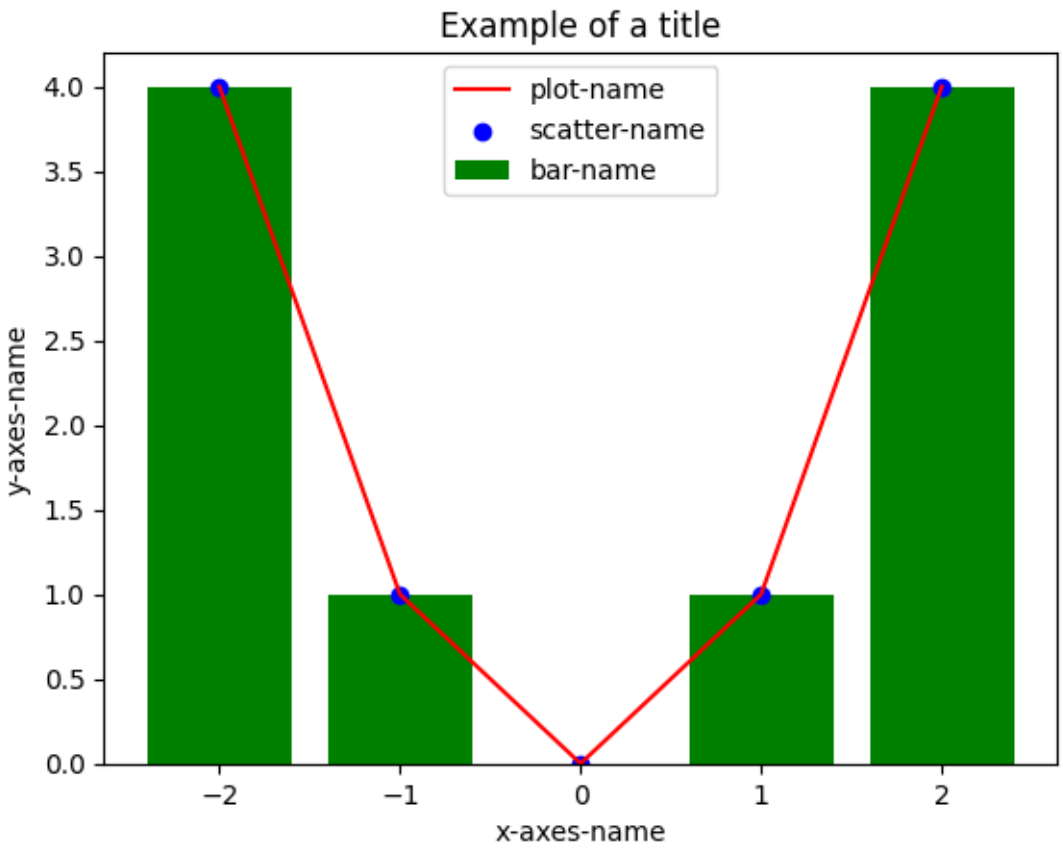
```
Out[6]: Text(0.5, 1.0, 'Example of a title')
```



We can plot multiple things on the same plot. Let's plot our values as a `plot`, a `bar` and a `scatter`.

```
In [7]: plt.plot(x_values, y_values, label="plot-name", color="red")
        plt.bar(x_values, y_values, label="bar-name", color="green")
        plt.scatter(x_values, y_values, label="scatter-name", color="blue")

        plt.legend()

        plt.xlabel("x-axes-name")
        plt.ylabel("y-axes-name")
        plt.title("Example of a title")
```

```
Out[7]: Text(0.5, 1.0, 'Example of a title')
```



`Matplotlib` also supports a lot of other types of plots, like bar charts, box plots, vector plots, and many more. If you can think of it, it's probably already implemented. If you need to use them, you will find lots of examples and help online.

15.2 Figure and Axes

15.2.1 Overview

In `matplotlib`, we have two main objects: the figure and the axes.

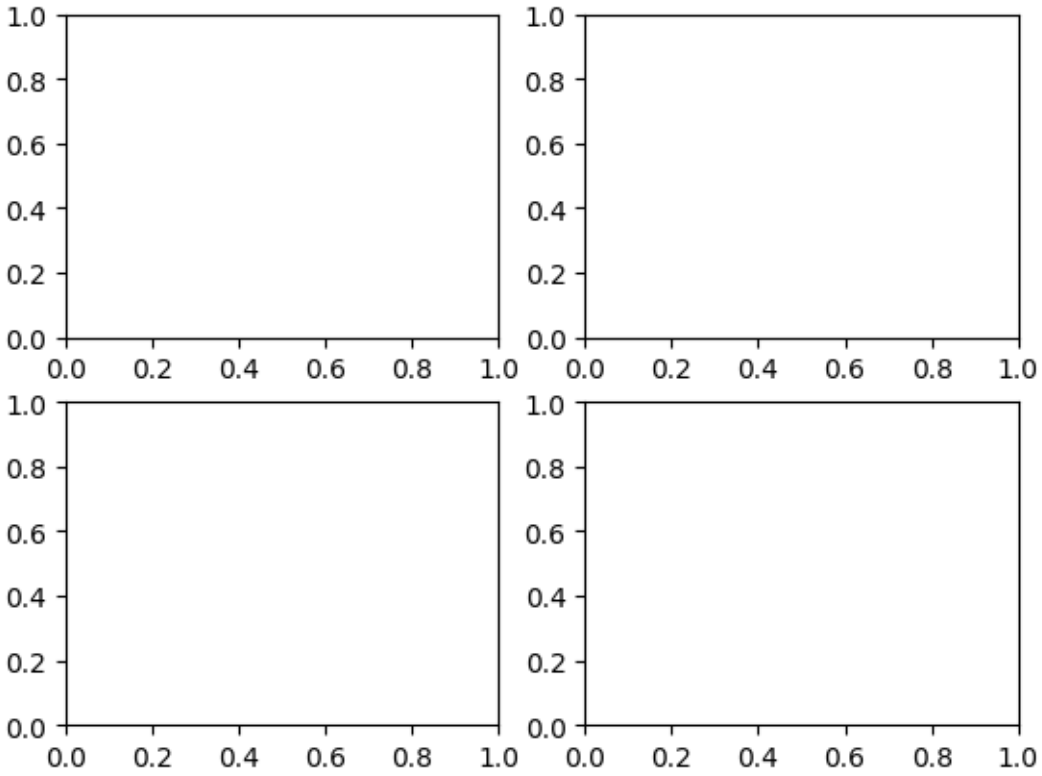
- Figure: The figure is the overall container for the entire visualization. It represents the entire image or canvas where plots, titles, legends, and other elements are drawn. A figure can hold one or more axes.
- Axes: The axes are the individual plots or graphing areas within a figure. Each axes object contains elements like the x-axis, y-axis, labels, and the actual data visualization. You can have multiple axes in a single figure to organize multiple plots.

A convenient way to arrange multiple axes in a grid-like structure within a figure are subplots. For example, you can create a grid of 2 rows and 2 columns of plots. Subplots allow you to visualize multiple datasets side by side in a structured manner.

15.2.2 Subplots

We can use `plt.subplots()` to return the "handles" for figure and the different plots in a list. Let's create a figure with four subplots. `plt.subplots(2, 2)` will create a 2x2 array of subplots - first argument is the number of rows, second is the number of columns.

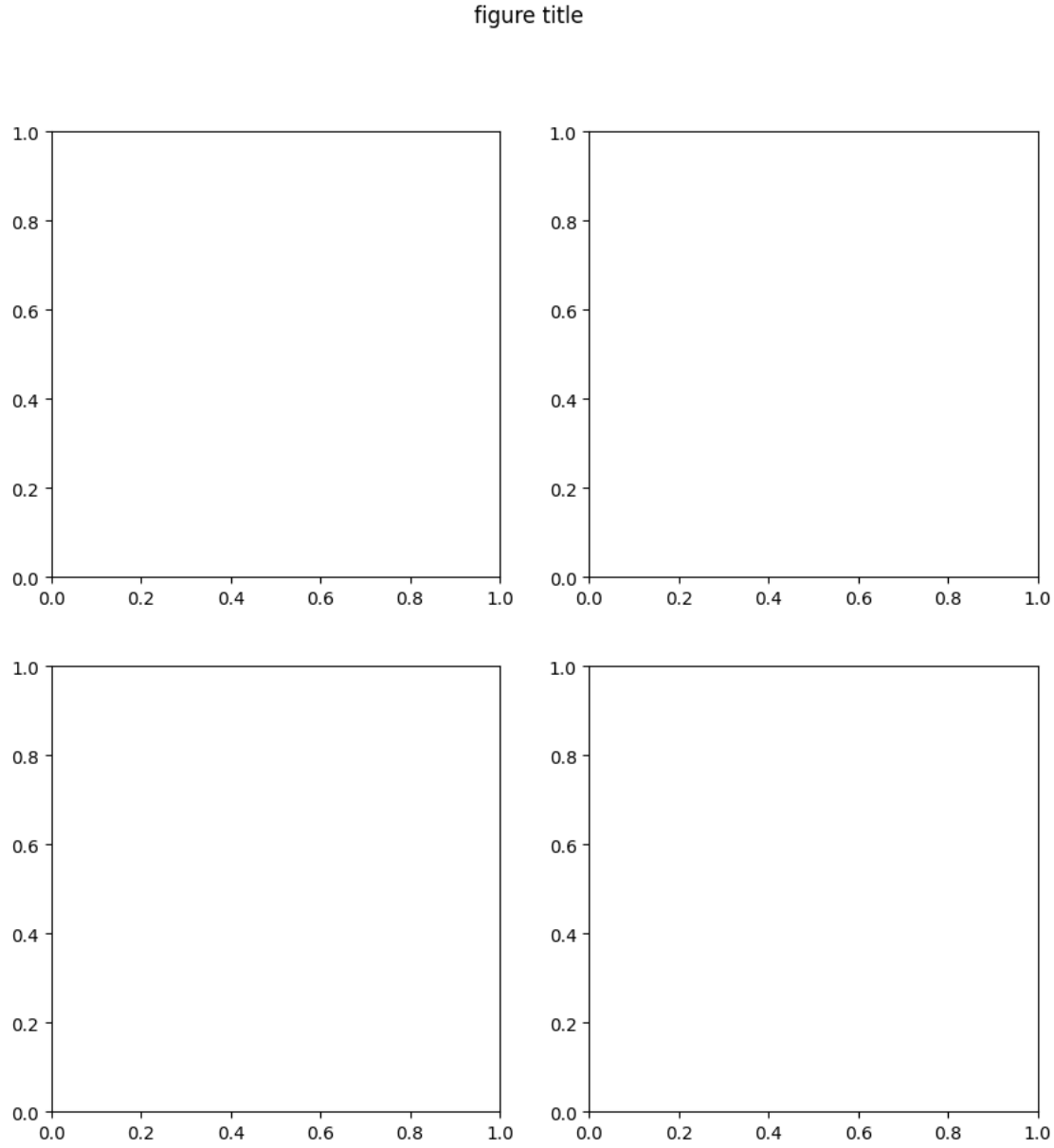
```
In [8]: fig, ax = plt.subplots(2, 2)  # nb_rows, nb_cols
```



We can now change/set the attributes of our figure by using the handle `fig`. E.g., we can set the size of the figure and we can set a title.

```
In [9]: fig, ax = plt.subplots(2, 2)
        fig.set_size_inches(10, 10)
        fig.suptitle("figure title")
```

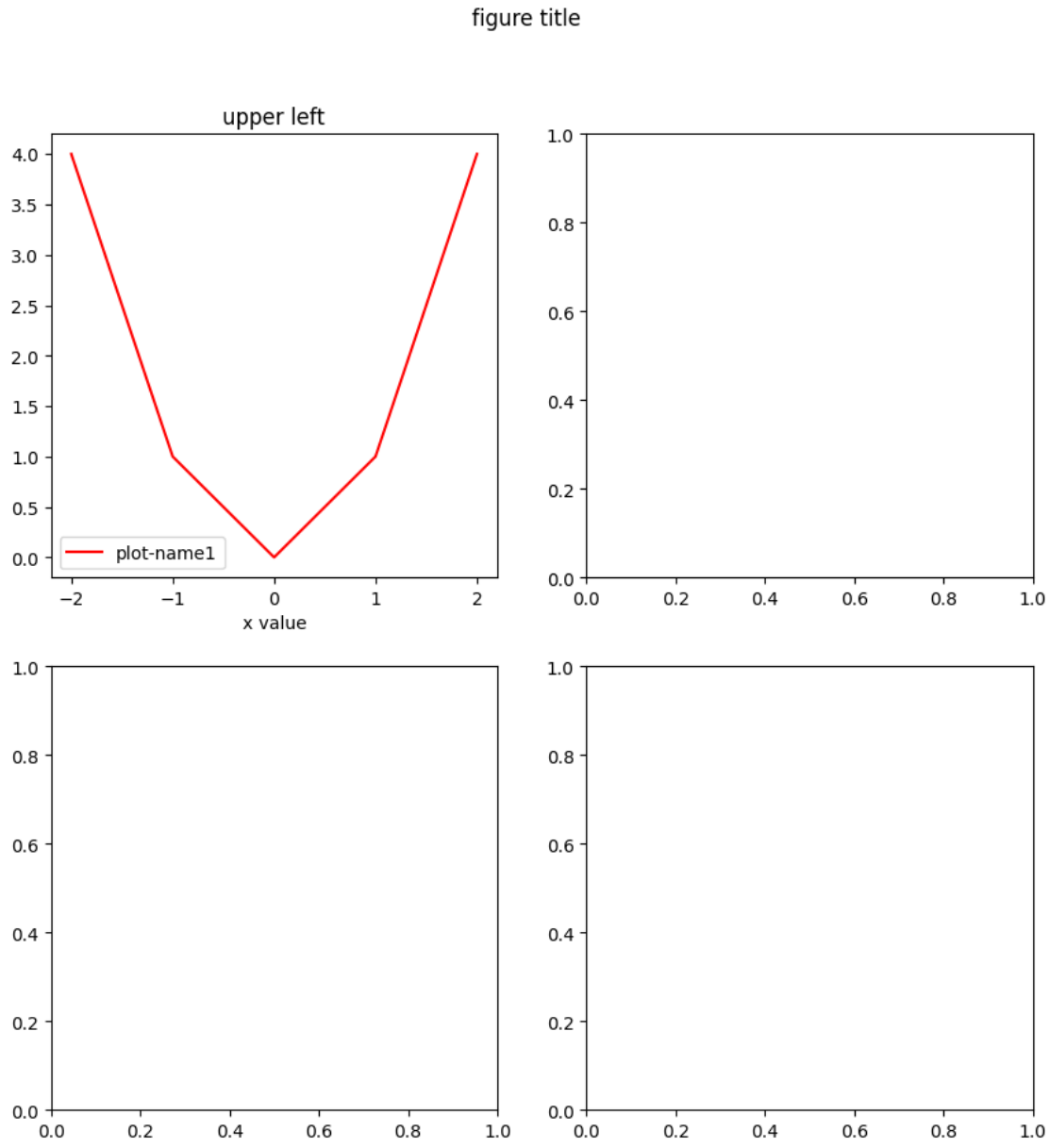
```
Out[9]: Text(0.5, 0.98, 'figure title')
```



To fill the individual axes, we use the handle `ax`. `ax[0, 0]` accesses the upper left subplot. In comparison to our small example from the beginning, the syntax changes a bit: - `plt.plot()` -> `ax[0, 0].plot()` - `plt.title()` -> `ax[0, 0].set_title()` - `plt.xlabel()` -> `ax.set_xlabel()` - ...

```
In [10]: fig, ax = plt.subplots(2, 2)

         fig.set_size_inches(10, 10)
         fig.suptitle("figure title")

         ax[0, 0].plot(x_values, y_values, label="plot-name1", color="red")
         ax[0, 0].set_title("upper left")
         ax[0, 0].set_xlabel("x value")
         ax[0, 0].legend()
```

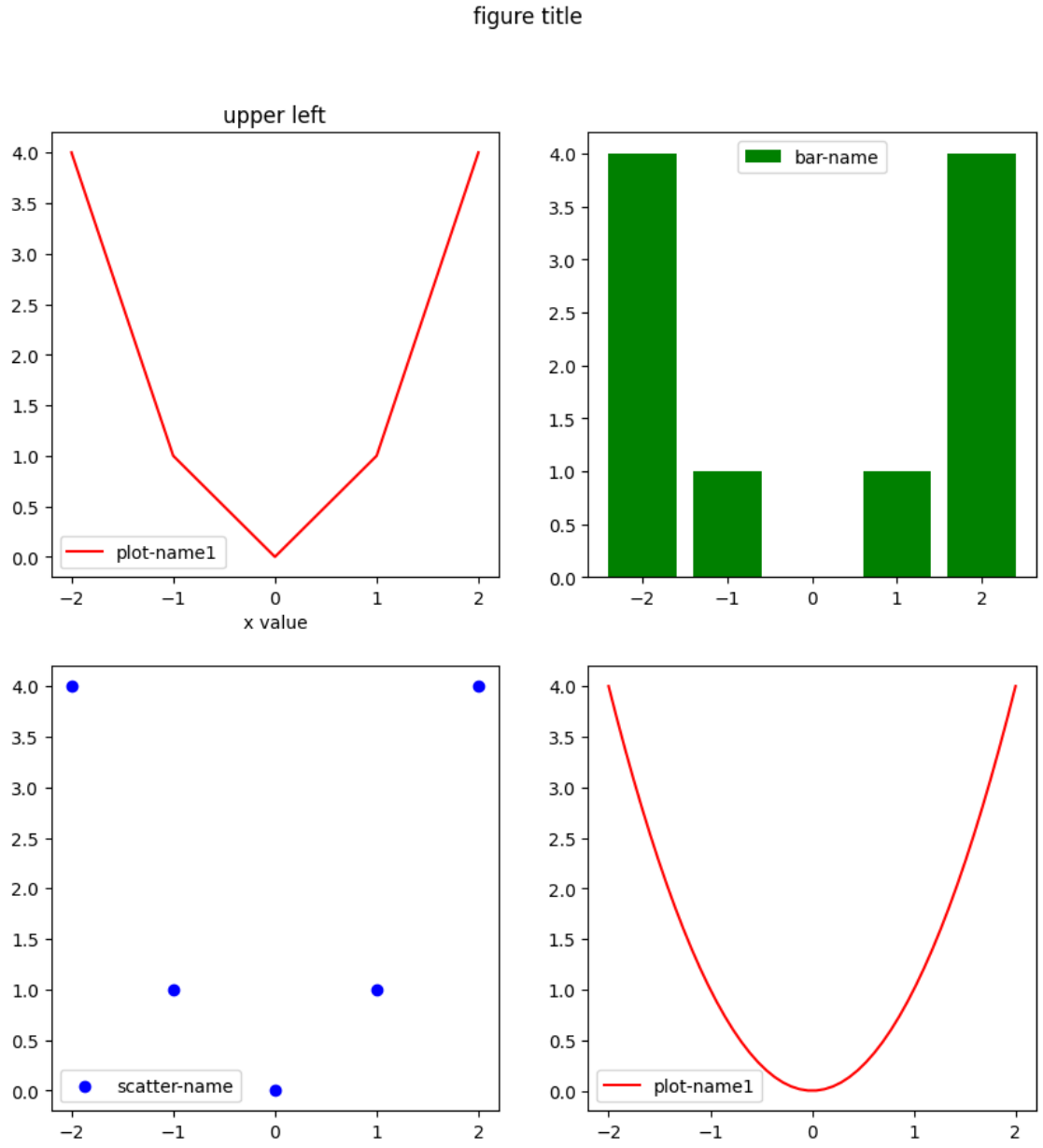
```
Out[10]: <matplotlib.legend.Legend at 0x7faf3ef486d0>
```



We can also fill the other three axes.

```
In [11]: fig, ax = plt.subplots(2, 2)

         fig.set_size_inches(10, 10)
         fig.suptitle("figure title")

         ax[0, 0].plot(x_values, y_values, label="plot-name1", color="red")
         ax[0, 0].set_title("upper left")
         ax[0, 0].set_xlabel("x value")
```

```
        # top right corner
        ax[0, 1].bar(x_values, y_values, label="bar-name", color="green")

        # bottom left corner
        ax[1, 0].scatter(x_values, y_values, label="scatter-name", color="blue")

        # bottom right corner
        ax[1, 1].plot(x_numpy, y_numpy, label="plot-name1", color="red")

        # iterate through each axes in the 2d array and add a legend
        # .flatten() converts the 2D array into a 1D array for
        # easier iteration
        for ax in ax.flatten():
            ax.legend()
```



It is often recommended to use `plt.subplots()` even if you only have one Axes, as it provides a consistent and flexible approach for managing plots, e.g. you can easily add plot and you use the same syntax.

```
In [12]: fig, ax = plt.subplots()
         ax.plot(x_numpy, y_numpy, label="plot-name1", color="red")
         fig.suptitle("Just one plot")
```

```
Out[12]: Text(0.5, 0.98, 'Just one plot')
```

15.2.3 GridSpec

`GridSpec` allows for more precise control over the layout of subplots in `matplotlib`. Unlike `plt.subplots()`, which creates a uniform grid of plots, `GridSpec` lets you define subplots that span multiple rows or columns, or have varying sizes. This is useful for creating complex or customized subplot arrangements.

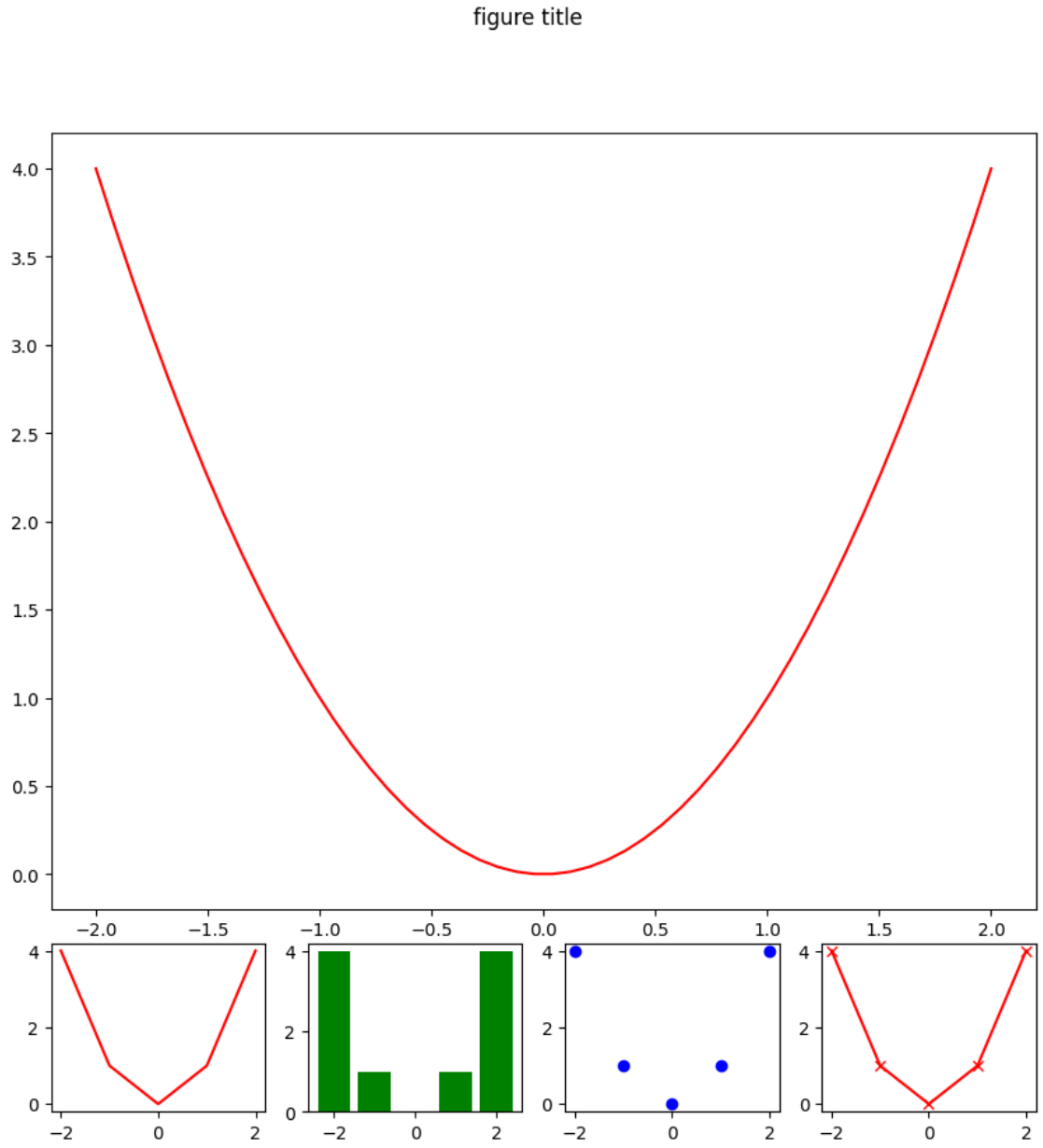
```
In [13]: import matplotlib.pyplot as plt  # as before

         # import gridspec as well
         import matplotlib.gridspec as gridspec
```

Let's create a figure with four small axis at the bottom and one large axis at the top.

```
In [14]: fig_grid = plt.figure(figsize=(10, 10))
         fig_grid.suptitle("figure title")

         # grid with 4 columns and 5 rows
         grid = gridspec.GridSpec(ncols=4, nrows=5, figure=fig_grid)

         # now we get the handler for the top plot
         # row 0 to 3, all columns
         top_plot = fig_grid.add_subplot(grid[0:4, :])
```

```
        top_plot.plot(x_numpy, y_numpy, label="plot-name", color="red")

        # add four subplots to the bottom row
        bottom_plots = [fig_grid.add_subplot(grid[-1, i]) for i in range(4)]

        # fill individual subplots at the bottom
        bottom_plots[0].plot(x_values, y_values, label="plot-name", color="red")
        bottom_plots[1].bar(x_values, y_values, label="bar-name", color="green")
        bottom_plots[2].scatter(x_values, y_values, label="scatter-name", color="blue")
        bottom_plots[3].plot(x_values, y_values, label="plot-name", marker="x",␣
↪color="red")
```

```
Out[14]: [<matplotlib.lines.Line2D at 0x7faf3f1a5050>]
```



15.3 Pandas: Data is Key

15.3.1 A small Data Frame

"pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language."

10 Minutes Introduction to pandas: https://pandas.pydata.org/docs/user_guide/10min.html

```
In [15]: import pandas as pd
```

We can only generate good graphs, if the data is good processed!
Create a simple data set from a dictionary containing products and sales quantities.

```
In [16]: santas = {"Type": ["Dark", "White", "Milk", "Hazelnut"], "Demand": [30, 40, 80,␣
 ↪30]}

         # convert to a pandas data frame


         df = pd.DataFrame(santas)
```

Print the dataframe.

```
In [17]: display(df)
```

```
       Type  Demand
0      Dark      30
1     White      40
2      Milk      80
3  Hazelnut      30
```

Why is this useful?
We can easily work with the data. For example, we can get the total sales over all products with `sum()`.

```
In [18]: total_sales = df["Demand"].sum()
         print(f" The total demand is: {total_sales}")
```

```
 The total demand is: 180
```

We can also sort all values by the index in descending order.

```
In [19]: df_sorted_ind = df.sort_index(ascending=False)
         display(df_sorted_ind)
```

```
       Type  Demand
3  Hazelnut      30
2      Milk      80
1     White      40
0      Dark      30
```

15.3.2 Modifying the data frame

We can add a new column to the data frame, e.g. the current inventory level of each product.

```
In [20]: inventory = [10, 40, 10, 10]

         # add new column to data frame
         df["Inventory"] = inventory

         display(df)
```

```
       Type  Demand  Inventory
0      Dark      30         10
1     White      40         40
2      Milk      80         10
3  Hazelnut      30         10
```

We can also add another row to the dataset. This can be done in multiple ways. In this case, we will use `concat`. At first, we create a second data set containing the new information.

```
In [21]: # create a new second data set
         df_new = pd.DataFrame(
             {"Type": ["Caramel", "Orange"], "Demand": [60, 40], "Inventory": [0, 0]}
         )
         display(df_new)
```

```
      Type  Demand  Inventory
0  Caramel      60          0
1   Orange      40          0
```

Now we can use `concat` to combine the datasets `df_new` and `df`.

```
In [22]: df_comb = pd.concat([df, df_new], ignore_index=True)
         display(df_comb)
```

```
       Type  Demand  Inventory
0      Dark      30         10
1     White      40         40
2      Milk      80         10
3  Hazelnut      30         10
4   Caramel      60          0
5    Orange      40          0
```

Why ignore_index? By default, `Pandas` keeps the original index values. However, working with duplicate index values can cause problems.

With `ignore_index=True`:

	Type	Demand	Inventory
0	Dark	30	10
1	White	40	40
2	Milk	80	10
3	Hazelnut	30	10
4	Caramel	60	0
5	Orange	40	0

With `ignore_index=False`:

	Type	Demand	Inventory
0	Dark	30	10
1	White	40	40
2	Milk	80	10
3	Hazelnut	30	10
0	Caramel	60	0
1	Orange	40	0

Now we want to compute the required production quantity. We need to add a new row `Production` in which we subtract the inventory level from the demand.

```
In [23]: df_comb["Production"] = df_comb["Demand"] - df_comb["Inventory"]
         display(df_comb)
```

	Type	Demand	Inventory	Production
0	Dark	30	10	20
1	White	40	40	0
2	Milk	80	10	70
3	Hazelnut	30	10	20
4	Caramel	60	0	60
5	Orange	40	0	40

Let's assume that we can sale a santa for 4€ (we neglect that different types may lead to different prices). We add a now column `Revenues` showing the revenues per type.

```
In [24]: revenue_per_santa = 4
         df_comb["Revenues"] = df_comb["Demand"] * revenue_per_santa
         display(df_comb)
```

	Type	Demand	Inventory	Production	Revenues
0	Dark	30	10	20	120
1	White	40	40	0	160
2	Milk	80	10	70	320
3	Hazelnut	30	10	20	120
4	Caramel	60	0	60	240
5	Orange	40	0	40	160

Let's calculate the total revenue with `sum()`.

```
In [25]: total_revenue = df_comb["Revenues"].sum()
         print(f"The total revenue is: {total_revenue}")
```

```
The total revenue is: 1120
```

We can rename a column name with `rename()`.

```
In [26]: df_renamed = df_comb.rename(columns={"Demand": "DEMAND", "Inventory":␣
 ↪"INVENTORY"})
         display(df_renamed)
```

```
       Type  DEMAND  INVENTORY  Production  Revenues
0      Dark      30         10          20       120
1     White      40         40           0       160
2      Milk      80         10          70       320
3  Hazelnut      30         10          20       120
4   Caramel      60          0          60       240
5    Orange      40          0          40       160
```

15.3.3 Sorting the data frame

We can sort the values in the data frame. Let's sort the types alphabetically with `sort_values()`.

```
In [27]: df_sorted = df_comb.sort_values("Type")
         display(df_sorted)
```

```
       Type  Demand  Inventory  Production  Revenues
4   Caramel      60          0          60       240
0      Dark      30         10          20       120
3  Hazelnut      30         10          20       120
2      Milk      80         10          70       320
5    Orange      40          0          40       160
1     White      40         40           0       160
```

We can also sort other columns. Let's sort the revenues with the highest revenue at the top using `sort_values()` and the option `ascending=False` .

```
In [28]: df_sorted_revenues = df_comb.sort_values("Revenues", ascending=False)
         display(df_sorted_revenues)
```

```
       Type  Demand  Inventory  Production  Revenues
2      Milk      80         10          70       320
4   Caramel      60          0          60       240
5    Orange      40          0          40       160
1     White      40         40           0       160
0      Dark      30         10          20       120
3  Hazelnut      30         10          20       120
```

We can change the sorting back to an index sorting with `sort_index()`.

```
In [29]: df_sorted_index = df_sorted_revenues.sort_index(ascending=True)
         display(df_sorted_index)
```

	Type	Demand	Inventory	Production	Revenues
0	Dark	30	10	20	120
1	White	40	40	0	160
2	Milk	80	10	70	320
3	Hazelnut	30	10	20	120
4	Caramel	60	0	60	240
5	Orange	40	0	40	160

15.3.4 Displaying the data frame and statistical information

Show only the first 5 rows of the data with `head()`.

```
In [30]: print("Head:")
         display(df_comb.head())
```

```
Head:
```

	Type	Demand	Inventory	Production	Revenues
0	Dark	30	10	20	120
1	White	40	40	0	160
2	Milk	80	10	70	320
3	Hazelnut	30	10	20	120
4	Caramel	60	0	60	240

Show only the last 5 rows of the data with `tail()`.

```
In [31]: print("Tail:")
         display(df_comb.tail())
```

```
Tail:
```

	Type	Demand	Inventory	Production	Revenues
1	White	40	40	0	160
2	Milk	80	10	70	320
3	Hazelnut	30	10	20	120
4	Caramel	60	0	60	240
5	Orange	40	0	40	160

Use `describe()` for some statistical information about each column in the data set.

```
In [32]: display(df_comb.describe())
```

	Demand	Inventory	Production	Revenues
count	6.000000	6.000000	6.000000	6.000000
mean	46.666667	11.666667	35.000000	186.666667

```
std     19.663842  14.719601   26.645825   78.655366
min     30.000000   0.000000    0.000000  120.000000
25%     32.500000   2.500000   20.000000  130.000000
50%     40.000000  10.000000   30.000000  160.000000
75%     55.000000  10.000000   55.000000  220.000000
max     80.000000  40.000000   70.000000  320.000000
```

We can use shape to get the number of rows and columns.

```
In [33]: rows, columns = df_comb.shape
         display(df_comb)
         print(f"There are {columns} columns and {rows} rows.")
```

	Type	Demand	Inventory	Production	Revenues
0	Dark	30	10	20	120
1	White	40	40	0	160
2	Milk	80	10	70	320
3	Hazelnut	30	10	20	120
4	Caramel	60	0	60	240
5	Orange	40	0	40	160

```
There are 5 columns and 6 rows.
```

Print the n random selected rows of the data with `sample()`.

```
In [34]: print("3 random rows:")
         display(df_comb.sample(3))
```

```
3 random rows:
```

	Type	Demand	Inventory	Production	Revenues
1	White	40	40	0	160
5	Orange	40	0	40	160
3	Hazelnut	30	10	20	120

15.3.5 Merging data frames

Assume that we have another dataset with the color of the wrapping. We would like to include the information within the other dateset. First, create the second dataset.

```
In [35]: # wrapping dataset
         df_wrapping = pd.DataFrame(
             {
                 # No information for the milk santa
                 "Type": ["Caramel", "Orange", "Hazelnut", "White", "Dark"],
                 "Wrapping_Color": ["brown", "orange", "green", "white", "blue"],
             }
         )
         display(df_wrapping)
```

```
       Type Wrapping_Color
0   Caramel          brown
1    Orange         orange
2  Hazelnut          green
3     White          white
4      Dark           blue
```

Use `merge()` to combine the two datasets. There are different options for merge. In this example, we will use `left` and `right` to demonstrate the differences.

`left`: Merge the two data frames based on the `Type` column of the **left** data frame (`df_comb`). As we have no wrapping color for the milk santa, we get an NaN at the cell.

```
In [36]: df_merged_left = pd.merge(df_comb, df_wrapping, how="left", on="Type")
         display(df_merged_left)
```

```
       Type  Demand  Inventory  Production  Revenues Wrapping_Color
0      Dark      30         10          20       120           blue
1     White      40         40           0       160          white
2      Milk      80         10          70       320            NaN
3  Hazelnut      30         10          20       120          green
4   Caramel      60          0          60       240          brown
5    Orange      40          0          40       160         orange
```

`right`: Merge the two data frames based on the `Type` column of the **right** data frame (`df_wrapping`). As we have no row for the milk santa in `df_wrapping`, we do not include the row of the milk santa from `df_comb`.

```
In [37]: df_merged_right = pd.merge(df_comb, df_wrapping, how="right", on="Type")
         display(df_merged_right)
```

```
       Type  Demand  Inventory  Production  Revenues Wrapping_Color
0   Caramel      60          0          60       240          brown
1    Orange      40          0          40       160         orange
2  Hazelnut      30         10          20       120          green
3     White      40         40           0       160          white
4      Dark      30         10          20       120           blue
```

15.3.6 Dataframe from a dictionary

```
In [38]: sales = {"product_1": 30, "product_2": 40, "product_3": 20}

         # convert to a pandas data frame
         df_dict = pd.DataFrame(sales.items(), columns=["Product", "SalesVolume"])
         display(df_dict)
```

```
     Product  SalesVolume
0  product_1           30
1  product_2           40
2  product_3           20
```

15.4 Using a pandas data frame for plotting

We now want to use data from a dataframe for plotting. Lets's consider an example, where we have four tasks. Every task has a start-time, a duration and the resource used for the task. The data is given as a dictionary.

Create a dataframe based on the data and generate a fixed color for each task with `seaborn`.

```
In [39]: # Import seaborn for easy color picking.
         import seaborn as sns
```

```
In [40]: data = {
             "Tasks": ["Task 1", "Task 2", "Task 3", "Task 4"],
             "Start Dates": [0, 4, 1, 3],
             "Durations": [3, 4, 2, 5],
             "Resources": ["A", "A", "B", "B"],
         }
         df = pd.DataFrame(data)
         display(df)
```

```
    Tasks  Start Dates  Durations Resources
0  Task 1            0          3         A
1  Task 2            4          4         A
2  Task 3            1          2         B
3  Task 4            3          5         B
```

```
In [41]: # We can add a new column to the data frame


         df["Color"] = list(sns.color_palette("hls", len(df)))


         display(df)
```

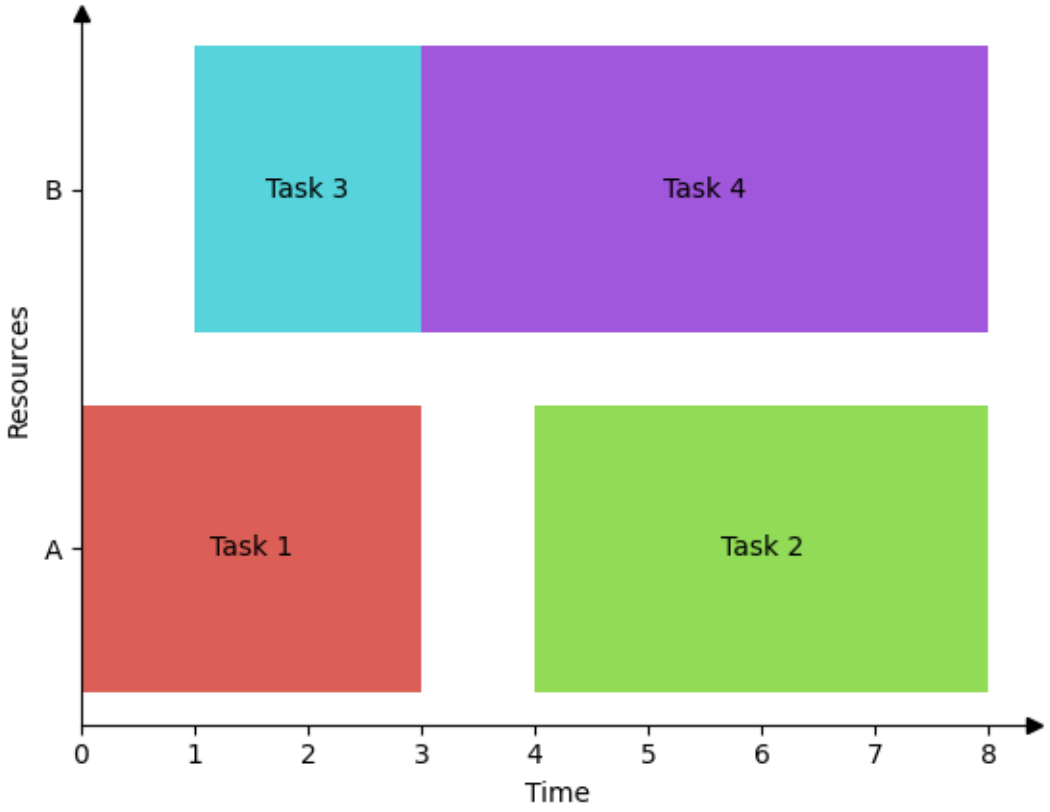
```
    Tasks  Start Dates  Durations Resources  \
0  Task 1            0          3         A
1  Task 2            4          4         A
2  Task 3            1          2         B
3  Task 4            3          5         B

                                        Color
0               (0.86, 0.3712, 0.33999999999999997)
1  (0.5688000000000001, 0.86, 0.33999999999999997)
2  (0.33999999999999997, 0.8287999999999999, 0.86)
3  (0.6311999999999998, 0.33999999999999997, 0.86)
```

Now we can use this dataframe and `matplotlib` to visualize the data. We will use an Gantt-Charts with the `barh()` function. Every task is a bar, the start of the bar is the beginning of the corresponding task and the length of the bar is the duration of the task. The y-axis is the resource and the x-axis is the time.

Create a figure showing the start time and duration of the tasks on the respective resources.

```
In [42]: fig, ax = plt.subplots()

         schedule = ax.barh(
```

```
            y=df["Resources"],  # values y
            width=df["Durations"],  # length of bars
            left=df["Start Dates"],  # start of bars
            label=df["Tasks"],
            color=df["Color"],
        )

        ax.bar_label(schedule, df["Tasks"], label_type="center", color="black")

        # set labels for the x and y axis
        ax.set_xlabel("Time")
        ax.set_ylabel("Resources")

        # arrows for the x and y axis
        # this will add an arrow to the right of the X axis
        ax.plot(1, 0, ">k", transform=ax.transAxes, clip_on=False)
        # this will add an arrow to the top of the Y axis look at marker documentation␣
↪for more options
        ax.plot(0, 1, "^k", transform=ax.transAxes, clip_on=False)
        # this will remove the top and right spines
        ax.spines[["right", "top"]].set_visible(False)
```



You can save figures using the `savefig()`.

```
In [43]: # create result folder if it does not exist
```

```
import os

path = "../results/lecture7"
if not os.path.exists(path):
    os.makedirs(path)

fig.savefig(f"{path}/schedule.png")
```

You can also show the figure in the notebook inside of markdown cells. For this, you need to use the **!** before the command.

Chapter 16

Matplotlib and Pandas II

16.1 Pandas

A helpful pandas cheat sheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

```
In [523]: import pandas as pd
```

16.1.1 Applying Changes to a data frame

There are multiple ways how we can apply changes to a data frame: We can create a new data frame or apply the changes to the same data frame. For the second option, we can use `inplace` or an assignment to the same data frame.

1. Create a new data frame

```
In [524]: sales = {"product_1": 30, "product_2": 40, "product_3": 20}

          df_dict = pd.DataFrame(sales.items(), columns=["Product", "SalesVolume"])

          df_sorted = df_dict.sort_values("SalesVolume")

          print('The original dict "df_dict"')
          display(df_dict)
          print('\nThe sorted DataFrame "df_sorted"')
          display(df_sorted)
```

```
The original dict "df_dict"
```

	Product	SalesVolume
0	product_1	30
1	product_2	40
2	product_3	20

```

The sorted DataFrame "df_sorted"
```

```
     Product  SalesVolume
2  product_3           20
0  product_1           30
1  product_2           40
```

2. Assignment to the same data frame.

```
In [525]: sales = {"product_1": 30, "product_2": 40, "product_3": 20}

          df_dict = pd.DataFrame(sales.items(), columns=["Product", "SalesVolume"])

          df_dict = df_dict.sort_values("SalesVolume")

          print('The original dict "df_dict"')
          display(df_dict)
```

```
The original dict "df_dict"
```

```
     Product  SalesVolume
2  product_3           20
0  product_1           30
1  product_2           40
```

3. Use `inplace=True`

```
In [526]: sales = {"product_1": 30, "product_2": 40, "product_3": 20}

          df_dict = pd.DataFrame(sales.items(), columns=["Product", "SalesVolume"])

          df_dict.sort_values("SalesVolume", inplace=True)

          print('The original dict "df_dict"')
          display(df_dict)
```

```
The original dict "df_dict"
```

```
     Product  SalesVolume
2  product_3           20
0  product_1           30
1  product_2           40
```

16.1.2 Using a custom index

Instead of using the default index, we can also set one of the columns as an index. This is especially useful when working with multiple data frames (see below).

```
In [527]: df_dict = df_dict.set_index("Product")
          display(df_dict)
```

```
           SalesVolume
Product
product_3           20
product_1           30
product_2           40
```

16.1.3 Loading data from csv

We have a csv dataset given containing the sales volume, cost and revenue for different year and products. We can load the data from a csv file with `read_csv()`.

```
In [528]: df_factory = pd.read_csv("../data/lecture_08/pandas_data/factory1.csv")

          display(df_factory)
```

	Year	Product	SalesVolume	Cost	Revenue
0	2021	Product_1	45	300	450
1	2021	Product_2	60	600	1200
2	2021	Product_3	100	100	500
3	2022	Product_1	45	300	450
4	2022	Product_2	65	650	1200
5	2022	Product_3	200	200	1000
6	2023	Product_1	45	350	450
7	2023	Product_2	90	950	2500
8	2023	Product_3	250	250	1250

16.1.4 Subsets of a data frame

In the last lecture, we learned how to use `sum()`. E.g., we can check the total sales volume.

```
In [529]: total_sales = df_factory["SalesVolume"].sum()
          print(f"The total sales volume is {total_sales}.")
```

```
The total sales volume is 900.
```

Now, we only want to get the total sales of `Product_2`. We can use `loc[]` for this.

```
In [530]: df_product_2 = df_factory.loc[df_factory["Product"] == "Product_2"]
          display(df_product_2)

          total_sales_prod_2 = df_product_2["SalesVolume"].sum()
          print(f"The total sales volume of product 2 is {total_sales_prod_2}.")
```

	Year	Product	SalesVolume	Cost	Revenue
1	2021	Product_2	60	600	1200
4	2022	Product_2	65	650	1200
7	2023	Product_2	90	950	2500

```
The total sales volume of product 2 is 215.
```

We can do the same thing with years. Get the total `Revenue` in the year 2022. Keep only the `Product` and the `Revenue` from the year 2022 in the data frame.

```
In [531]: # create a new data frame only containing the year
          df_year_2022 = df_factory.loc[df_factory["Year"] == 2022][["Product",␣
 ↪"Revenue"]]
          display(df_year_2022)

          total_revenue_2022 = df_year_2022["Revenue"].sum()
          print(f"The total revenue in 2022 is {total_revenue_2022}.")
```

	Product	Revenue
3	Product_1	450
4	Product_2	1200
5	Product_3	1000

```
The total revenue in 2022 is 2650.
```

Let's compute the profit per unit and write it in a new column of the same DataFrame.

```
In [532]: df_factory["Profit_per_unit"] = (
              (df_factory["Revenue"] - df_factory["Cost"]) / df_factory["SalesVolume"]
          ).round(2)

          display(df_factory)
```

	Year	Product	SalesVolume	Cost	Revenue	Profit_per_unit
0	2021	Product_1	45	300	450	3.33
1	2021	Product_2	60	600	1200	10.00
2	2021	Product_3	100	100	500	4.00
3	2022	Product_1	45	300	450	3.33
4	2022	Product_2	65	650	1200	8.46
5	2022	Product_3	200	200	1000	4.00
6	2023	Product_1	45	350	450	2.22
7	2023	Product_2	90	950	2500	17.22
8	2023	Product_3	250	250	1250	4.00

Create a data frame with the two most recent years and a profit per unit ≥ 4.

```
In [533]: df_profit = df_factory.loc[
              (df_factory["Year"] >= 2022) & (df_factory["Profit_per_unit"] >= 4)
          ]
          display(df_profit)
```

	Year	Product	SalesVolume	Cost	Revenue	Profit_per_unit
4	2022	Product_2	65	650	1200	8.46
5	2022	Product_3	200	200	1000	4.00
7	2023	Product_2	90	950	2500	17.22
8	2023	Product_3	250	250	1250	4.00

We can also access the data frame by the row locations with `iloc[]`. - `loc[]`: gets a row by its label - `iloc[]`: gets a row by its location/position

```
In [534]: display(df_factory.iloc[2:5])

          df_sorted_factory = df_factory.sort_values("Product")

          display(df_sorted_factory.iloc[2:5])
```

	Year	Product	SalesVolume	Cost	Revenue	Profit_per_unit
2	2021	Product_3	100	100	500	4.00
3	2022	Product_1	45	300	450	3.33
4	2022	Product_2	65	650	1200	8.46

	Year	Product	SalesVolume	Cost	Revenue	Profit_per_unit
6	2023	Product_1	45	350	450	2.22
1	2021	Product_2	60	600	1200	10.00
4	2022	Product_2	65	650	1200	8.46

16.1.5 Reshaping the data frame

We have another data set with the revenues of a factory over multiple years for different products given. We want to create a data frame from the dictionary and compute the mean revenue of `Product_1`.

```
In [535]: revenues = {
              (2019, "Produkt_1"): 284,
              (2019, "Produkt_2"): 193.34535,
              (2019, "Produkt_3"): 77,
              (2019, "Produkt_4"): 132,
              (2020, "Produkt_1"): 102,
              (2020, "Produkt_2"): 356,
              (2020, "Produkt_3"): 268,
              (2020, "Produkt_4"): 189.787,
              (2021, "Produkt_1"): 274,
              (2021, "Produkt_2"): 162,
              (2021, "Produkt_3"): 320,
              (2021, "Produkt_4"): 378,
              (2021, "Produkt_5"): 95.00000009,
              (2022, "Produkt_1"): 99,
              (2022, "Produkt_2"): 211,
              (2022, "Produkt_3"): 354,
              (2022, "Produkt_4"): 201,
              (2022, "Produkt_5"): 383.76,
          }
```

We start with creating a data frame from the dictionary.

```
In [536]: df_revenues = pd.DataFrame(revenues.items(), columns=["Key", "Revenue"])
          display(df_revenues)
```

	Key	Revenue
0	(2019, Produkt_1)	284.00000
1	(2019, Produkt_2)	193.34535
2	(2019, Produkt_3)	77.00000

```
3   (2019, Produkt_4)  132.00000
4   (2020, Produkt_1)  102.00000
5   (2020, Produkt_2)  356.00000
6   (2020, Produkt_3)  268.00000
7   (2020, Produkt_4)  189.78700
8   (2021, Produkt_1)  274.00000
9   (2021, Produkt_2)  162.00000
10  (2021, Produkt_3)  320.00000
11  (2021, Produkt_4)  378.00000
12  (2021, Produkt_5)   95.00000
13  (2022, Produkt_1)   99.00000
14  (2022, Produkt_2)  211.00000
15  (2022, Produkt_3)  354.00000
16  (2022, Produkt_4)  201.00000
17  (2022, Produkt_5)  383.76000
```

We split the column `Key` in two new columns `Year` and `Product` using `tolist()`.

```
In [537]: df_revenues[["Year", "Product"]] = pd.DataFrame(df_revenues["Key"].tolist())
          display(df_revenues)
```

```
                  Key    Revenue  Year    Product
0   (2019, Produkt_1)  284.00000  2019  Produkt_1
1   (2019, Produkt_2)  193.34535  2019  Produkt_2
2   (2019, Produkt_3)   77.00000  2019  Produkt_3
3   (2019, Produkt_4)  132.00000  2019  Produkt_4
4   (2020, Produkt_1)  102.00000  2020  Produkt_1
5   (2020, Produkt_2)  356.00000  2020  Produkt_2
6   (2020, Produkt_3)  268.00000  2020  Produkt_3
7   (2020, Produkt_4)  189.78700  2020  Produkt_4
8   (2021, Produkt_1)  274.00000  2021  Produkt_1
9   (2021, Produkt_2)  162.00000  2021  Produkt_2
10  (2021, Produkt_3)  320.00000  2021  Produkt_3
11  (2021, Produkt_4)  378.00000  2021  Produkt_4
12  (2021, Produkt_5)   95.00000  2021  Produkt_5
13  (2022, Produkt_1)   99.00000  2022  Produkt_1
14  (2022, Produkt_2)  211.00000  2022  Produkt_2
15  (2022, Produkt_3)  354.00000  2022  Produkt_3
16  (2022, Produkt_4)  201.00000  2022  Produkt_4
17  (2022, Produkt_5)  383.76000  2022  Produkt_5
```

We can now delete the column `Key`. The axis can either be the row or the column of a DataFrame. To access the row use `axis=0` and `axis=1` for a column.

```
In [538]: df_revenues = df_revenues.drop("Key", axis=1)
          display(df_revenues)
```

```
     Revenue  Year    Product
0  284.00000  2019  Produkt_1
1  193.34535  2019  Produkt_2
2   77.00000  2019  Produkt_3
3  132.00000  2019  Produkt_4
4  102.00000  2020  Produkt_1
```

```
5   356.00000  2020  Produkt_2
6   268.00000  2020  Produkt_3
7   189.78700  2020  Produkt_4
8   274.00000  2021  Produkt_1
9   162.00000  2021  Produkt_2
10  320.00000  2021  Produkt_3
11  378.00000  2021  Produkt_4
12   95.00000  2021  Produkt_5
13   99.00000  2022  Produkt_1
14  211.00000  2022  Produkt_2
15  354.00000  2022  Produkt_3
16  201.00000  2022  Produkt_4
17  383.76000  2022  Produkt_5
```

We can use `pivot()` to reshape the data. We want to use the Year as the index and the products as the columns. The revenues should be the values of the combinations.

```
In [539]: df_revenues = df_revenues.pivot(index="Year", columns="Product",␣
 ↪values="Revenue")
          display(df_revenues)
```

Product	Produkt_1	Produkt_2	Produkt_3	Produkt_4	Produkt_5
Year					
2019	284.0	193.34535	77.0	132.000	NaN
2020	102.0	356.00000	268.0	189.787	NaN
2021	274.0	162.00000	320.0	378.000	95.00
2022	99.0	211.00000	354.0	201.000	383.76

We can also modify our data. E.g., let's replace all `NaN` with 0 and round all values to two decimal points.

```
In [540]: # round values
          df_revenues = df_revenues.round(2)
          display(df_revenues)
```

Product	Produkt_1	Produkt_2	Produkt_3	Produkt_4	Produkt_5
Year					
2019	284.0	193.35	77.0	132.00	NaN
2020	102.0	356.00	268.0	189.79	NaN
2021	274.0	162.00	320.0	378.00	95.00
2022	99.0	211.00	354.0	201.00	383.76

```
In [541]: # replace nan with 0
          df_revenues = df_revenues.fillna(0)
          display(df_revenues)
```

Product	Produkt_1	Produkt_2	Produkt_3	Produkt_4	Produkt_5
Year					
2019	284.0	193.35	77.0	132.00	0.00
2020	102.0	356.00	268.0	189.79	0.00
2021	274.0	162.00	320.0	378.00	95.00
2022	99.0	211.00	354.0	201.00	383.76

16.1.6 Multiple Data frames

We have a dictionary with students, their id and their age given. Convert the dictionary to a pandas data frame and set the index to `student_id`.

```
In [542]: students = {
              "student_id": [1004, 1005, 1006, 1007],
              "student_name": ["Mustafa", "Anna", "Lara", "Jan"],
              "age": [20, 23, 22, 21],
          }

          df_students = pd.DataFrame(students)
          df_students = df_students.set_index("student_id")
          display(df_students)
```

```
            student_name  age
student_id
1004             Mustafa   20
1005                Anna   23
1006                Lara   22
1007                 Jan   21
```

We have the exam results from three exams given. Each exam result is stored in a different csv file. Create a data frame for each of those three exam results and add a row containing the total points. Store the data frames in a dictionary.

```
In [543]: # number of exam written in the class
          nb_exams = 3
          # dictionary for storing the exam results
          exam_results = {}

          for i in range(1, nb_exams + 1):

              exam_name = f"exam_{i}"

              # load the results from the csv file and create a data frame
              df_exam = pd.read_csv(f"../data/lecture_08/pandas_data/{exam_name}.csv")

              # set the index of the data frame to the student number
              df_exam = df_exam.set_index("student_id")

              # compute a new column with the total points of each student per exam,
              # axis = 1: We apply the sum to each column
              df_exam["total_points"] = df_exam.sum(axis=1)
              display(df_exam)

              # save the data frame in the dictionary
              exam_results[exam_name] = df_exam
```

```
            exam_question_1  exam_question_2  exam_question_3  total_points
student_id
1004                      3                4                3            10
1005                      3                2                2             7
```

```
1006                    1                 1                 0             2
1007                    2                 0                 3             5
```

```
            exam_question_1   exam_question_2   exam_question_3  \
student_id
1005                      5                 4                 5
1007                      4                 2                 3
1006                      2                 3                 0
1004                      4                 4                 3

            exam_question_4   exam_question_5  total_points
student_id
1005                      5                 3            22
1007                      3                 0            12
1006                      0                 0             5
1004                      2                 3            16
```

```
            exam_question_1   exam_question_2  total_points
student_id
1006                      5                 7            12
1004                     10                 9            19
1005                      7                 8            15
```

We can now access the individual data frames, e.g. the data frame for `exam_1`.

```
In [544]: display(exam_results["exam_1"])
```

```
            exam_question_1   exam_question_2   exam_question_3  total_points
student_id
1004                      3                 4                 3            10
1005                      3                 2                 2             7
1006                      1                 1                 0             2
1007                      2                 0                 3             5
```

Print the points of the student `Lara` in the first exam.

```
In [545]: df = exam_results["exam_1"]
          df_lara = df.loc[df_students["student_name"] == "Lara"]
          display(df_lara)
```

```
            exam_question_1   exam_question_2   exam_question_3  total_points
student_id
1006                      1                 1                 0             2
```

Compute the total number of points for each student and add the result to the data frame `df_students`.

```
In [546]: df_students["points"] = 0
          display(df_students)

          for df in exam_results.values():
              df_students["points"] = df_students["points"].add(df["total_points"],␣
 ↪fill_value=0)

          display(df_students)
```

```
           student_name  age  points
student_id
1004            Mustafa   20       0
1005               Anna   23       0
1006               Lara   22       0
1007                Jan   21       0
```

```
           student_name  age  points
student_id
1004            Mustafa   20    45.0
1005               Anna   23    44.0
1006               Lara   22    19.0
1007                Jan   21    17.0
```

16.1.7 Use `apply()` to apply a function to the data frame

A data set with products current prices and an increase in percent is given. Merge the two DataFrames and compute the new price in the same DataFrame.

```
In [547]: price = {
              "product_1": 30,
              "product_2": 40,
              "product_3": 20,
              "product_4": 30,
              "product_5": 20,
          }

          increase = {
              "product_1": 3,
              "product_2": 4,
              "product_3": 5,
              "product_4": 7,
              "product_5": 3,
          }

          df_price = pd.DataFrame(price.items(), columns=["Product", "Price"])
          df_increase = pd.DataFrame(increase.items(), columns=["Product",␣
 ↪"PercentageIncrease"])

          # Merge the two DataFrames on the 'Product' column
          df_combined = pd.merge(df_price, df_increase, on="Product")

          display(df_combined)
```

```
     Product  Price  PercentageIncrease
0  product_1     30                   3
1  product_2     40                   4
2  product_3     20                   5
3  product_4     30                   7
4  product_5     20                   3
```

To fill the new column we write a function with the desired row as output.

```
In [548]: def increase_price(row):
              return row["Price"] * (1 + row["PercentageIncrease"] / 100)
```

Next we use `apply()` to run the function for every row in our DataFrame.

```
In [549]: df_combined["NewPrice"] = df_combined.apply(increase_price, axis=1)

          display(df_combined)
```

	Product	Price	PercentageIncrease	NewPrice
0	product_1	30	3	30.9
1	product_2	40	4	41.6
2	product_3	20	5	21.0
3	product_4	30	7	32.1
4	product_5	20	3	20.6

16.2 Matplotlib with function and models

16.2.1 Splitted plotting of the vrp

In the last lecture, we learned how to plot data with matplotlib. Now we want to imbed the plot in a function and use it to plot the results of a model from gurobipy.

Meme:

Frist we take our vrp model and instance generator. And modify it slightly, so that the coordinates are also saved in the instance.

```
In [550]: import matplotlib.pyplot as plt
          import gurobipy as gp
          from gurobipy import GRB
          import pickle
          import seaborn as sns
```

```
import sys

sys.path.append("../data/lecture_08")

import instance_module_vrp as im
from build_vrp import build_vrp
```

Next we create our instances which we want to solve and analyse.

```
In [551]: import os

          if not os.path.exists("../data/vrp_data"):
              os.makedirs("../data/vrp_data")

          ins = im.generate_instances(nb_instances=10, capacity=5, nb_locations=10)
```

To create reusable code, we can create a function. The function takes the instance as an input and returns the model and the variables. How we create the model will be explained in lecture 10. Only the output of the model is relevant for this lecture.

```
In [552]: def solve_vrp(env, instance):
              # define an empty model
              vrp = gp.Model(env=env)

              # build the model
              X, Y = build_vrp(vrp=vrp, instance=instance)

              vrp.optimize()

              return vrp, X, Y
```

For every part, we want to plot, we create a own function, this helps us to keep the code clean and readable. We start with the customers. All of the information we need is stored in the instance.

```
In [553]: def plot_coordinates(ax, instance):
              # plot the coordinates of the depot
              x, y = instance.coordinates[instance.depot]
              ax.plot([x], [y], marker="o", color="red", alpha=1)
              ax.text(x, y + 0.1, "Depot", ha="center", fontsize=8)

              # Add the locations of all customers
              for l in instance.locations[1:]:
                  x, y = instance.coordinates[l]
                  ax.plot([x], [y], marker="o", color="black", alpha=1)
                  ax.text(x, y + 0.1, l, ha="center", fontsize=8)
```

Now we want to plot also the routes of the vehicles. Therefore, we need to add the following code as a own function.

`instance.tours` are all vehicles in the instance. One vehicle has a maximum of one tour.

```
In [554]: def plot_vrp_solution(ax, instance, vrp, X_vars):
              # create a color for every tour
              colorpalett = list(sns.color_palette("hls", len(instance.tours)))
```

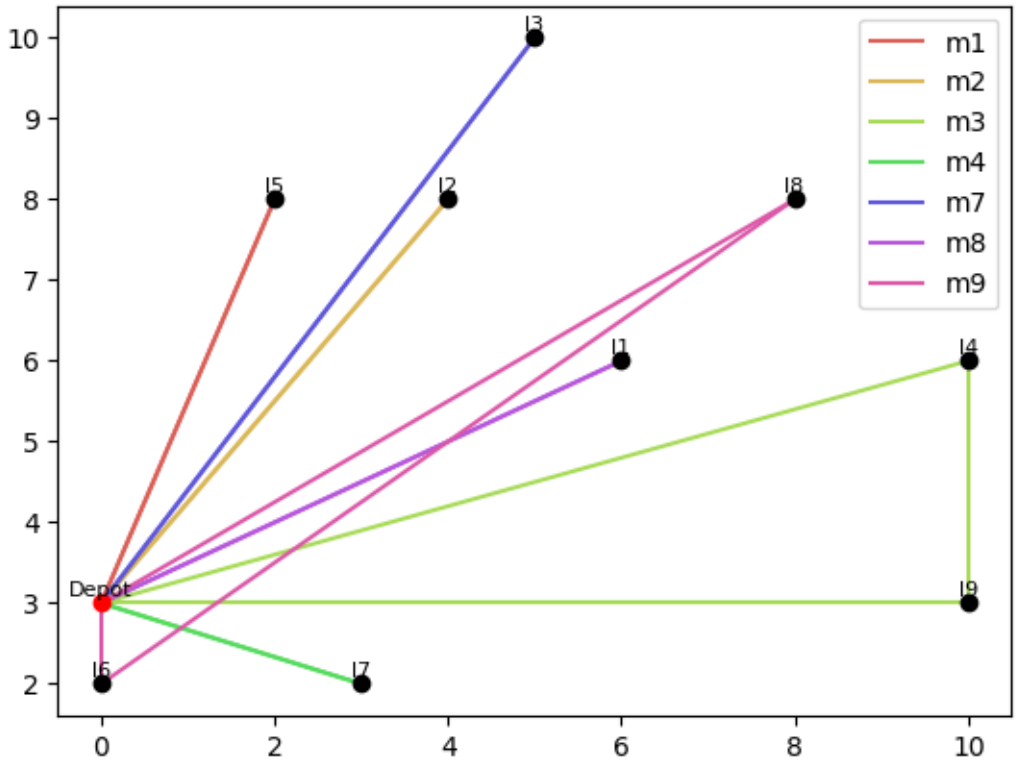
```
            colors = {tour: color for tour, color in zip(instance.tours, colorpalett)}

            # Check whether we found a solution or not
            if vrp.Status == GRB.OPTIMAL or vrp.Status == GRB.TIME_LIMIT:
                # Iterate thorugh all vehicles
                for tour in instance.tours:
                    # Iterate through all combinations of locations
                    for origin in instance.locations:
                        for destination in instance.locations:
                            label = None
                            # Set the label for the tour
                            if origin == instance.depot:
                                label = tour
                            # Check if the vehicle travels from origin to destination
                            if X_vars[origin, destination, tour].X > 0.5:
                                x_values, y_values = zip(
                                    *[
                                        instance.coordinates[origin],
                                        instance.coordinates[destination],
                                    ]
                                )
                                ax.plot(x_values, y_values, color=colors[tour],␣
↪label=label)
            else:
                print("No solution found.")
            ax.legend()

            print(x_values, y_values)
```

Now we can call the functions and plot the results. To save some time later, we define a function that calls all the other functions.

```
In [555]: def solve_and_plot_vrp(ax, instance):
            # create the environment
            env = gp.Env()
            env.setParam("OutputFlag", 0)
            env.setParam("timelimit", 30)

            # solve the model
            vrp_model, X, _ = solve_vrp(env=env, instance=instance)

            # plot the solution
            plot_vrp_solution(ax=ax, instance=instance, vrp=vrp_model, X_vars=X)

            # plot the coordinates of the customers
            plot_coordinates(ax, instance)
```

```
In [556]: ins = pickle.load(open("../data/vrp_data/i10_b5_2.vrp", "rb"))

          fig_vrp, ax_vrp = plt.subplots()

          solve_and_plot_vrp(ax=ax_vrp, instance=ins)
```

```
Restricted license - for non-production use only - expires 2026-11-23
(8, 0) (8, 2)
```



16.2.2 Plotting with geopandas

Now this is nice. But it can be nicer. Lets generate instances, wich are inspired from real world data. We use `geopandas` to get the coordinates of a city. We can use this to generate instances. And plot them.

`geopandas` is a library to work with geodata. It is based on `pandas` and uses `matplotlib` for plotting. First we need data. We can download data from the internet. or we can use packages, which provide the datasets. Then we can load it with `geopandas`. Here is a good tutorial for germany data.

```
In [557]: import geopandas as gpd
```

We use the shapefile-format (shp), we need to unpack the .zip into a folder and then load the folder via `geopandas`. We can use the `read_file()` function to load the data. We can also use the `head()` function to get a first impression of the data. We now have a pandas-dataframe. In this dataframe are various data, but the most important is the geometry. This is a polygon, which describes the shape (in this case the shape of one plz). We can plot it with `geopandas`.

```
In [558]: plz_shape_df = gpd.read_file(
              "../data/lecture_08/plz-5stellig.shp", dtype={"plz": str}, encoding="utf-8"
          )

          plz_shape_df.head()
```

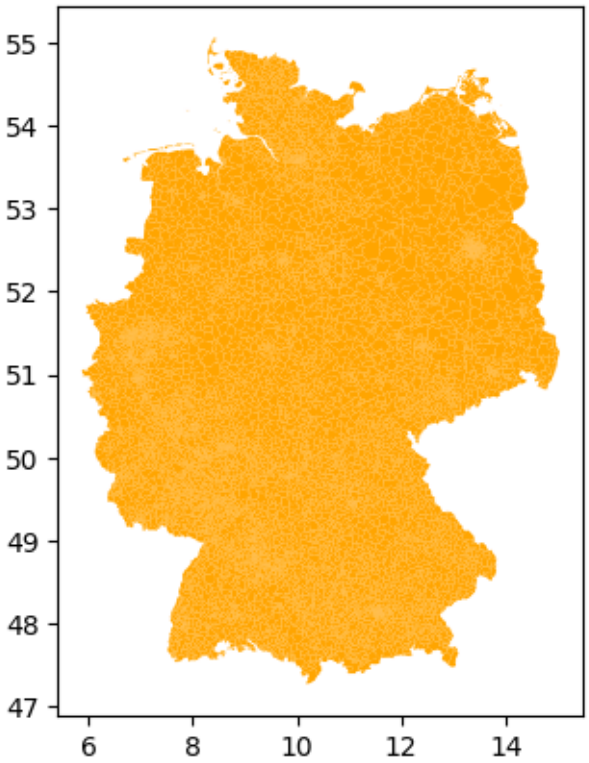
```
/usr/local/lib/python3.11/dist-packages/pyogrio/raw.py:196: RuntimeWarning: driver ESRI␣
↪Shapefile does not support open option DTYPE
  return ogr_read(
```

Out[558]:

	plz	note	einwohner	qkm	geometry
0	64743	Situation unklar, evtl. haben die Häuser Marba...	3	0.082066	POLYGON ((8.98124 49.60761, 8.98312 49.60748, ...
1	81248	81248 München	121	1.984763	POLYGON ((11.39468 48.14729, 11.3949 48.1478, ...
2	60315	60315 Frankfurt am Main (FOUR)	0	0.017285	POLYGON ((8.67254 50.11264, 8.67258 50.11265, ...
3	99331	99331 Geratal	4523	20.207080	POLYGON ((10.79153 50.69477, 10.79178 50.69819...
4	60312	60312 Frankfurt am Main (Omniturm)	0	0.001829	POLYGON ((8.67262 50.11164, 8.67311 50.11182, ...

```
In [559]: fig_germany, ax_germany = plt.subplots()
          plz_shape_df.plot(ax=ax_germany, color="orange")
```

Out[559]: <Axes: >

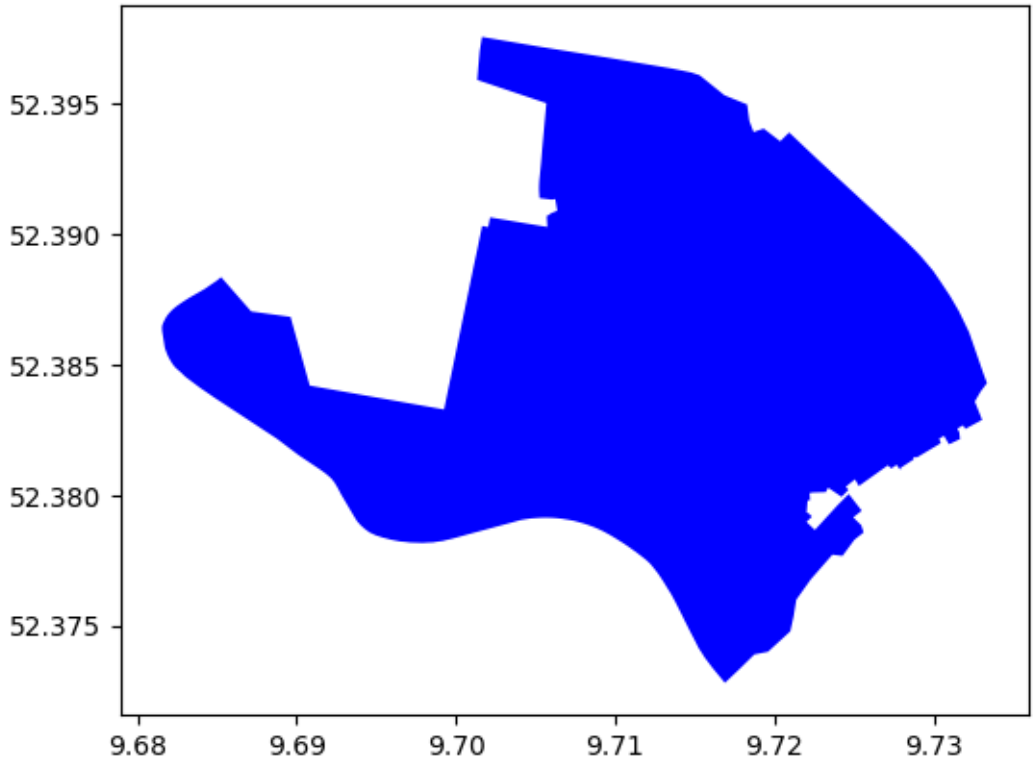


We can also only display a part of the data, for example the shape of plz=30167.

```
In [560]: fig_30167, ax_30167 = plt.subplots()
          plz_30167 = plz_shape_df.loc[plz_shape_df["plz"] == "30167"]
          print(plz_30167.head())
          plz_30167.plot(ax=ax_30167, color="blue")
```

```
        plz            note  einwohner       qkm  \
2668  30167  30167 Hannover      17777  4.343813

                                                 geometry
2668  POLYGON ((9.68154 52.38642, 9.68154 52.38645, ...
```

```
Out[560]: <Axes: >
```

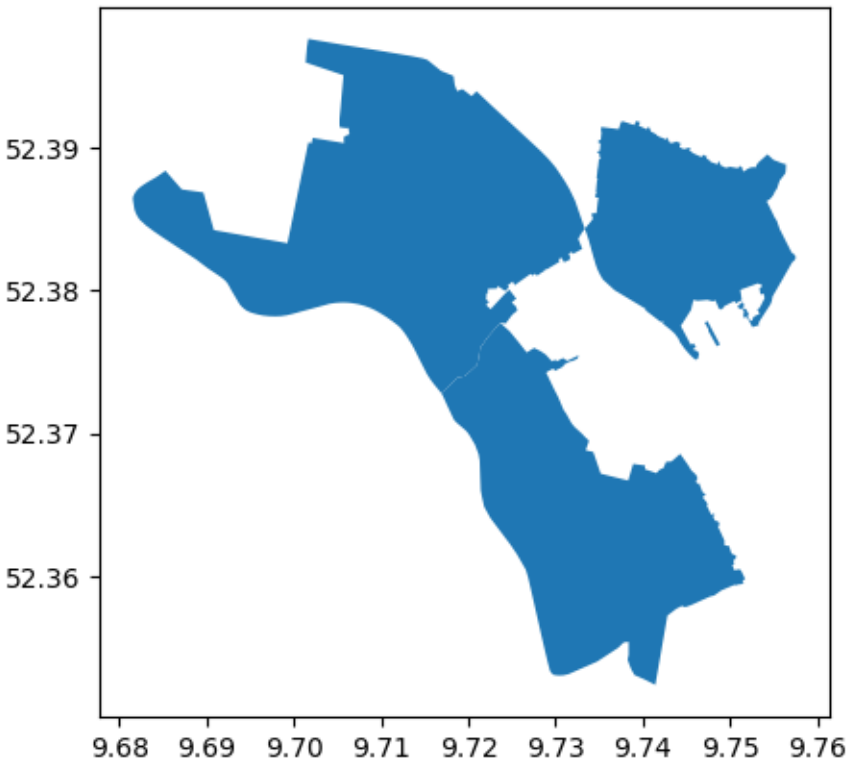



```
In [561]: fig_multiple_plz, ax_multiple_plz = plt.subplots()
          plz_list = ["30167", "30169", "30161"]

          plz_collection = plz_shape_df.loc[plz_shape_df["plz"].isin(plz_list)]
          display(plz_collection)

          plz_collection.plot(ax=ax_multiple_plz)
```

```
        plz            note  einwohner       qkm  \
2480  30161  30161 Hannover      25332  1.789440
```

```
2668  30167  30167 Hannover      17777  4.343813
2681  30169  30169 Hannover       9987  3.017907

                                                 geometry
2480  MULTIPOLYGON (((9.73372 52.38457, 9.73383 52.3...
2668  POLYGON ((9.68154 52.38642, 9.68154 52.38645, ...
2681  POLYGON ((9.7169 52.3728, 9.71783 52.37334, 9...
```

```
Out[561]: <Axes: >
```

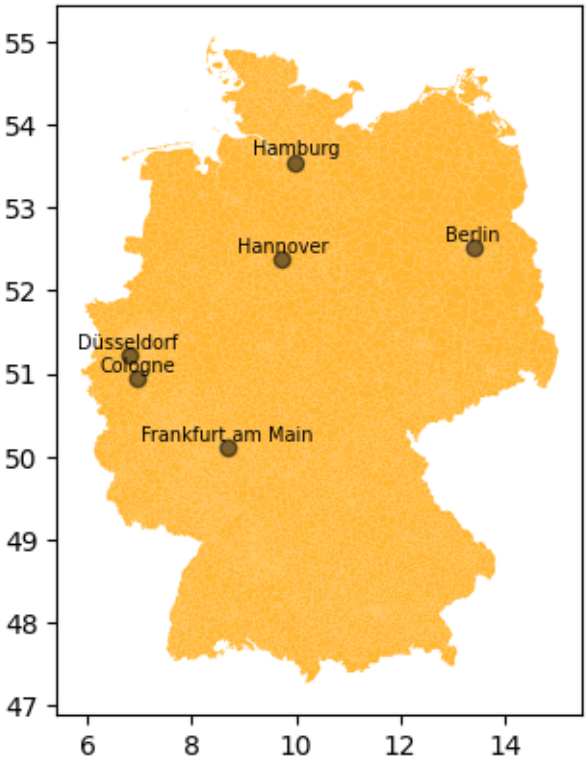


Add some cities with there coordinates to the data.

```
In [562]: # Get lat and lng of Germany's main cities.
          top_cities = {
              "Hannover": (9.73322, 52.37052),
              "Berlin": (13.404954, 52.520008),
              "Cologne": (6.953101, 50.935173),
              "Düsseldorf": (6.782048, 51.227144),
              "Frankfurt am Main": (8.682127, 50.110924),
              "Hamburg": (9.993682, 53.551086),
          }
```

```
In [563]: fig_ger_citys, ax_ger_citys = plt.subplots()

          plz_shape_df.plot(ax=ax_ger_citys, color="orange", alpha=0.8)
```

```
        # Plot cities.
        for city in top_cities.keys():
            x_coordinate, y_coordinate = top_cities[city]
            # Plot city location centroid.
            ax_ger_citys.plot(x_coordinate, y_coordinate, marker="o", color="black",␣
↪alpha=0.5)
            # Plot city name.
            ax_ger_citys.text(
                x=x_coordinate,
                # Add small shift to avoid overlap with point.
                y=y_coordinate + 0.08,
                s=city,
                fontsize=7,
                ha="center",
            )
```



16.2.3 Real Instance from Data

Now we want to generate a "real" instance. We can use the coordinates of the cities to generate a instance. We leave the rest to our instance-generator.

```
In [564]: real_instance = im.VrpData(
              instance_number=0,
              capacity=7,
              nb_locations=None,
```

```
            locations=list(top_cities.keys()),
            distance=None,
            coordinates=top_cities,
            transport_units=None,
        )
        print(real_instance)
```

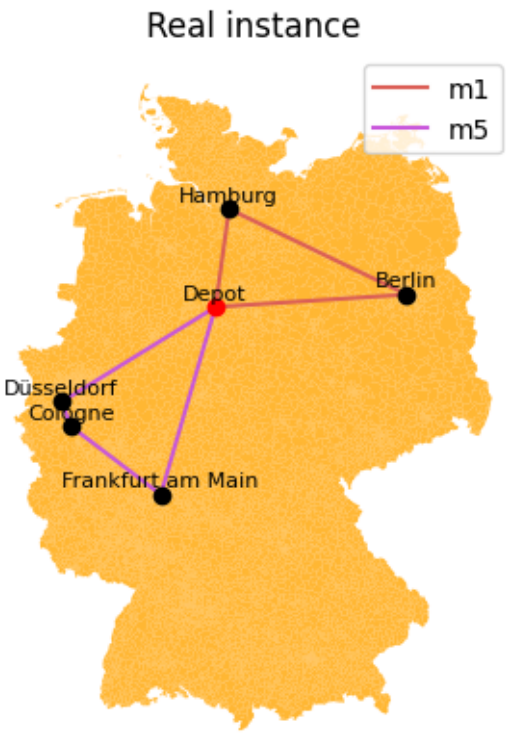
```
VRP(i6_b7_0, 7, 6, ['Hannover', 'Berlin', 'Cologne', 'Düsseldorf', 'Frankfurt am Main',␣
↪'Hamburg'], {('Hannover', 'Hannover'): 0.0, ('Hannover', 'Berlin'): 3.67, ('Berlin',␣
↪'Hannover'): 3.67, ('Hannover', 'Cologne'): 3.13, ('Cologne', 'Hannover'): 3.13,␣
↪('Hannover', 'Düsseldorf'): 3.16, ('Düsseldorf', 'Hannover'): 3.16, ('Hannover',␣
↪'Frankfurt am Main'): 2.49, ('Frankfurt am Main', 'Hannover'): 2.49, ('Hannover',␣
↪'Hamburg'): 1.21, ('Hamburg', 'Hannover'): 1.21, ('Berlin', 'Berlin'): 0.0, ('Berlin',␣
↪'Cologne'): 6.64, ('Cologne', 'Berlin'): 6.64, ('Berlin', 'Düsseldorf'): 6.75,␣
↪('Düsseldorf', 'Berlin'): 6.75, ('Berlin', 'Frankfurt am Main'): 5.3, ('Frankfurt am␣
↪Main', 'Berlin'): 5.3, ('Berlin', 'Hamburg'): 3.56, ('Hamburg', 'Berlin'): 3.56,␣
↪('Cologne', 'Cologne'): 0.0, ('Cologne', 'Düsseldorf'): 0.34, ('Düsseldorf',␣
↪'Cologne'): 0.34, ('Cologne', 'Frankfurt am Main'): 1.92, ('Frankfurt am Main',␣
↪'Cologne'): 1.92, ('Cologne', 'Hamburg'): 4.01, ('Hamburg', 'Cologne'): 4.01,␣
↪('Düsseldorf', 'Düsseldorf'): 0.0, ('Düsseldorf', 'Frankfurt am Main'): 2.2,␣
↪('Frankfurt am Main', 'Düsseldorf'): 2.2, ('Düsseldorf', 'Hamburg'): 3.96, ('Hamburg',␣
↪'Düsseldorf'): 3.96, ('Frankfurt am Main', 'Frankfurt am Main'): 0.0, ('Frankfurt am␣
↪Main', 'Hamburg'): 3.68, ('Hamburg', 'Frankfurt am Main'): 3.68, ('Hamburg',␣
↪'Hamburg'): 0.0}, {'Hannover': (9.73322, 52.37052), 'Berlin': (13.404954, 52.520008),␣
↪'Cologne': (6.953101, 50.935173), 'Düsseldorf': (6.782048, 51.227144), 'Frankfurt am␣
↪Main': (8.682127, 50.110924), 'Hamburg': (9.993682, 53.551086)}, {'Berlin': 3,␣
↪'Cologne': 3, 'Düsseldorf': 2, 'Frankfurt am Main': 2, 'Hamburg': 2, 'Hannover': 0})
```

To combine our results of the model with the background of the map, we just need to plot the map first. Our coordinates are already in the right format.

```
In [565]: fig_real, ax_real = plt.subplots()
          plz_shape_df.plot(ax=ax_real, color="orange", alpha=0.8)
          solve_and_plot_vrp(ax=ax_real, instance=real_instance)

          ax_real.set_title("Real instance")
          ax_real.spines[:].set_visible(False)
          ax_real.set_xticks([])
          ax_real.set_yticks([])
```

```
Restricted license - for non-production use only - expires 2026-11-23
(8.682127, 9.73322) (50.110924, 52.37052)
```

```
Out[565]: []
```



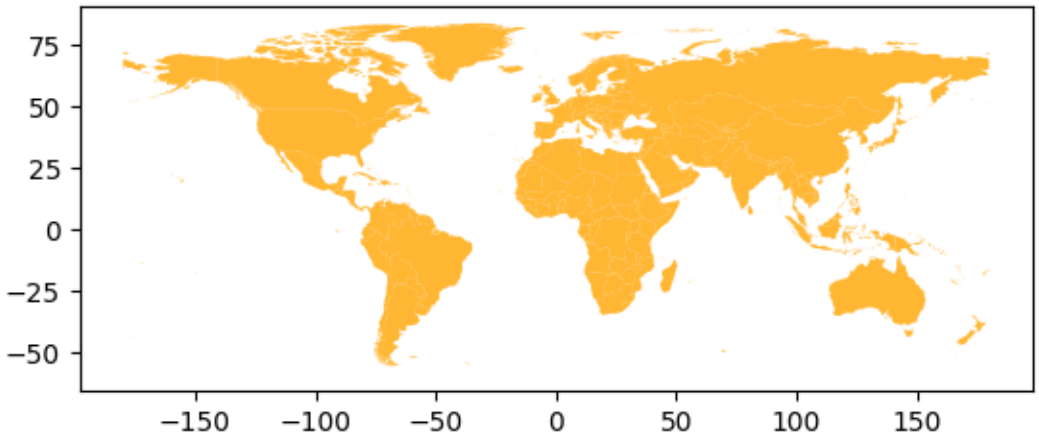
we can use the same methods, to work with other data-sets. Here for example the world administrative boundaries. We use only shapefiles in this course, because, they are the most common format (but also the oldest). But there are also other formats, like geojson, or geopackage.

```
In [566]: world_countries_df = gpd.read_file("../data/lecture_08/
 ↪world-administrative-boundaries")

          display(world_countries_df.head())

          fig_world, ax_world = plt.subplots()
          world_countries_df.plot(ax=ax_world, color="orange", alpha=0.8)
```

```
   iso3                status color_code                     name continent  \
0   MNP          US Territory        USA  Northern Mariana Islands   Oceania
1  None  Sovereignty unsettled        RUS             Kuril Islands      Asia
2   FRA          Member State        FRA                    France    Europe
3   SRB          Member State        SRB                    Serbia    Europe
4   URY          Member State        URY                   Uruguay  Americas

            region iso_3166_1_               french_shor  \
0       Micronesia          MP  Northern Mariana Islands
1     Eastern Asia        None             Kuril Islands
2   Western Europe          FR                    France
3  Southern Europe          RS                    Serbie
4    South America          UY                   Uruguay
```

```
                                                geometry
0  MULTIPOLYGON (((145.63331 14.91236, 145.62412 ...
1  MULTIPOLYGON (((146.68274 43.70777, 146.66664 ...
2  MULTIPOLYGON (((9.4475 42.68305, 9.45014 42.63...
3  POLYGON ((20.26102 46.11485, 20.31403 46.06986...
4  POLYGON ((-53.3743 -33.74067, -53.39917 -33.75...
```

```
Out[566]: <Axes: >
```




```
In [567]: world_germany = world_countries_df.loc[world_countries_df["iso3"] == "DEU"]

          fig_real_v2, ax_real_v2 = plt.subplots()
          world_germany.plot(ax=ax_real_v2, color="orange", alpha=0.8)
          solve_and_plot_vrp(ax=ax_real_v2, instance=real_instance)

          ax_real_v2.set_title("Real instance - version 2")
          ax_real_v2.spines[:].set_visible(False)
          ax_real_v2.set_xticks([])
          ax_real_v2.set_yticks([])
```

```
Restricted license - for non-production use only - expires 2026-11-23
```

```
(8.682127, 9.73322) (50.110924, 52.37052)
```

```
Out[567]: []
```


Real instance - version 2
m1
m5
Hamburg
Berlin
Depot
Düsseldorf
Cologne
Frankfurt am Main

Chapter 17

KI

17.1 Setup Copilot

GitHub Copilot can help you to speed up your coding process by providing real-time suggestions based on the context of your code. It uses AI to suggest common patterns and solutions, allowing you to focus more on the logic and less on repetitive tasks.

You need tp get a free account for students. An instruction can be found at https://techcommunity.microsoft.com/blog/educatordeveloperblog/step-by-step-setting-up-github-student-and-github-copilot-as-an-authenticated-st/3736279

In addition, you need the **VSCode GitHub Copilot** and **GitHub Copilot Chat** extensions.

Afterwards, you can login to Copilot via VS Code (see https://code.visualstudio.com/docs/copilot/setup).

We see in the status bar if copilot is active:

17.2 Using Copilot

```
1  # create a simple pandas dataframe
```

If activated, Copilot will give you suggestions for your code:
The suggestion will be shown in light grey. You can accept the suggestion by using tab.

However, if you get to little information, the suggestion will not help you to much:

```
1  # create a simple web server that listens on port 8080
```

Python

With good comments, you can even get a suggestion for a full (or even multiple) lines of code:

```
1  santas = {"Type": ["Dark", "White", "Milk", "Hazelnut"], "Demand": [30, 40, 80, 30]}
2  df = pd.DataFrame(santas)
3
4  # Calculate the mean demand
5  mean_demand = df["Demand"].mean()
6
7
```

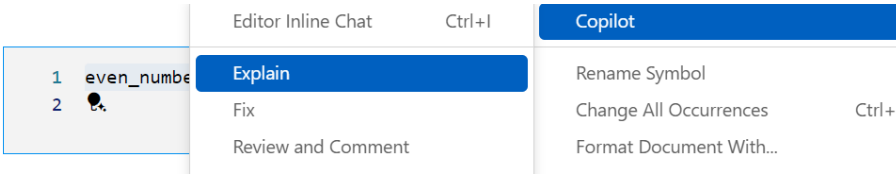
Python

Try it out

yourself:

```
In [ ]: import pandas as pd

        santas = {"Type": ["Dark", "White", "Milk", "Hazelnut"], "Demand": [30, 40, 80,␣
↪30]}
        df = pd.DataFrame(santas)

        # Comment
```

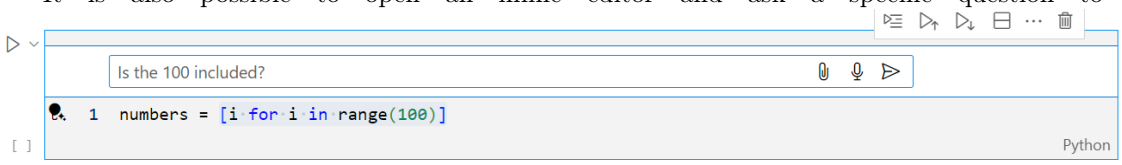
```
3
6
```



We can also ask to explain parts of the code:
Try it out yourself:

```
In [ ]: even_numbers = [i for i in range(100) if i % 2 == 0]
```

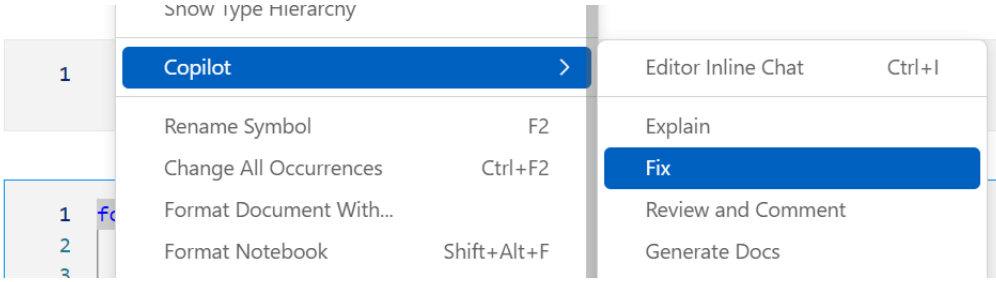
It is also possible to open an inline editor and ask a specific question to the code:



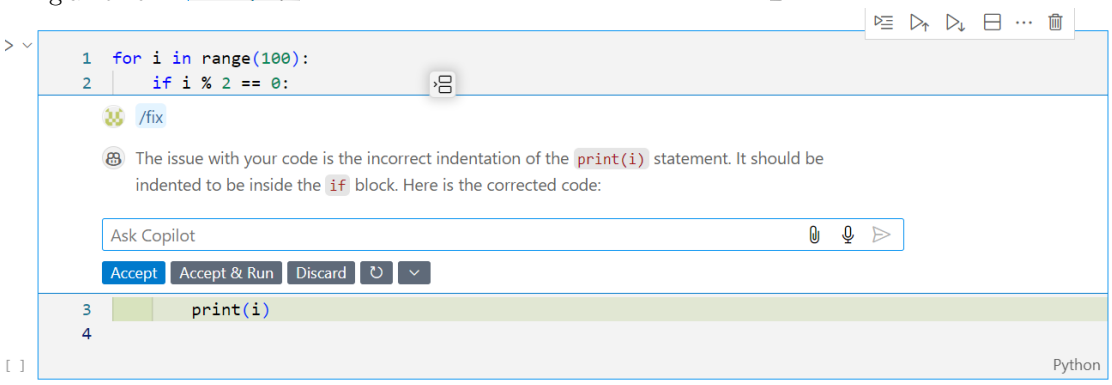
Try it out yourself:

```
In [ ]: numbers = [i for i in range(100)]
```

And we can ask for fixing an error:



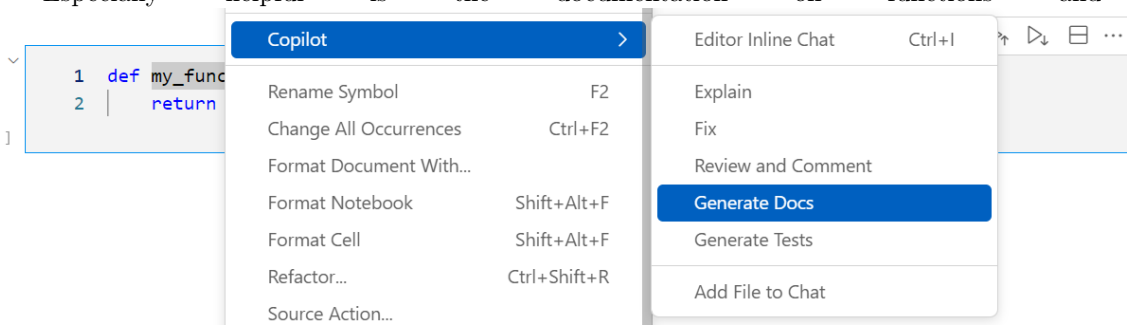
Which results into:



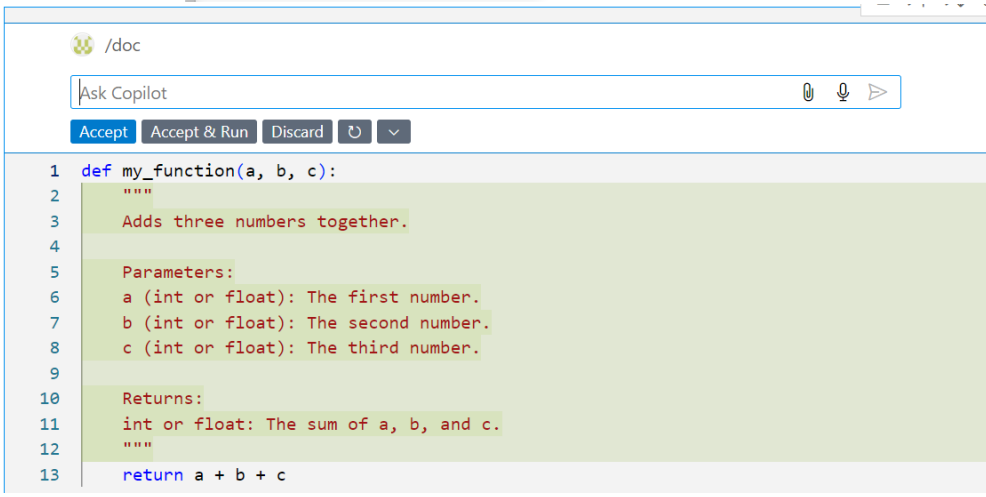
Try it out yourself:

```
In [ ]: for i in range(100):
            ...
```

Especially helpful is the documentation on functions and classes:



Copilot will make you a suggestion:



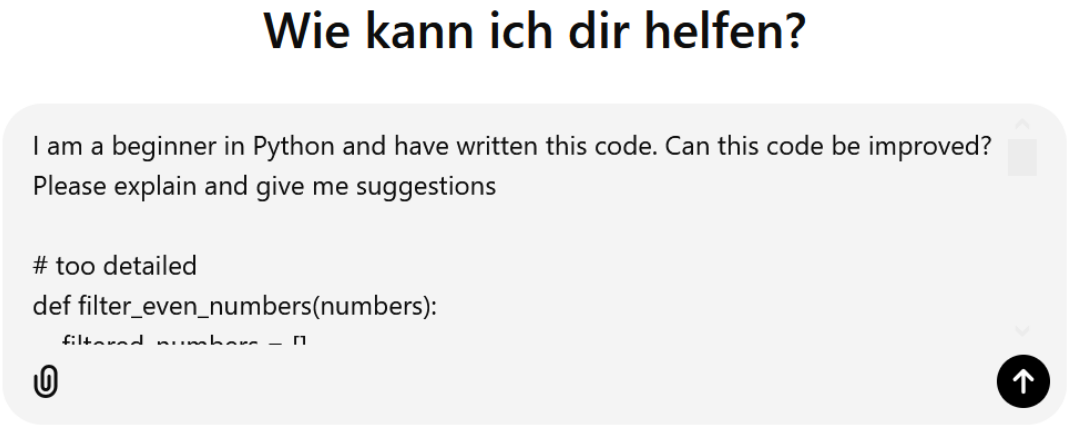
Try it out yourself:

```
In [ ]: def my_function(a, b, c):
            return a + b + c
```

Keep in mind that this will not always cover everything and that you might need to add or change something. However, it will give you a very good starting point.

17.3 ChatGPT

You can also you use ChatGPT for improving your programming skills. E.g. you could ask for an improvement:



You will get a detailed description of the improvements. Try it out yourself:

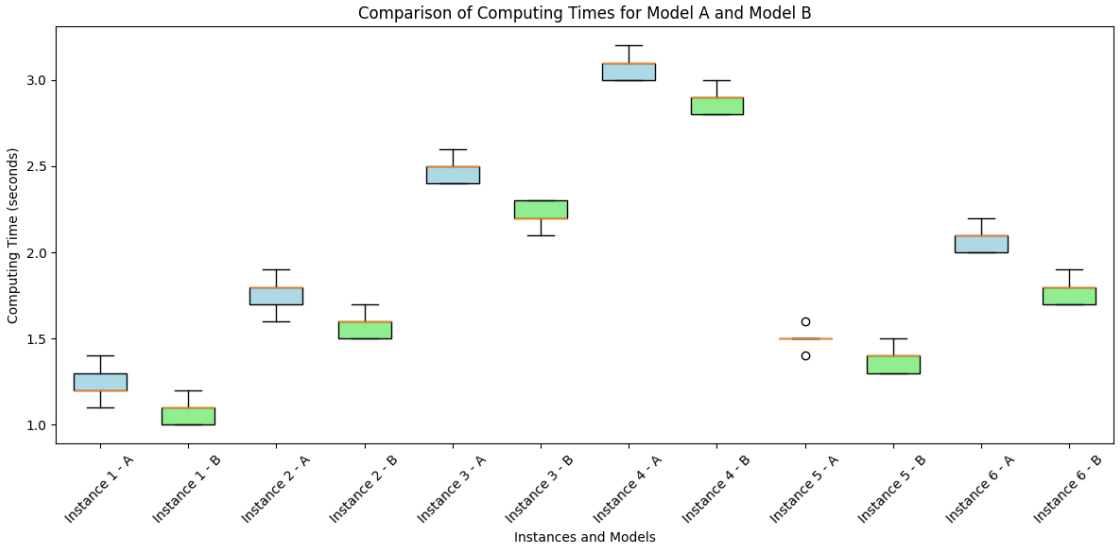
```
In [ ]: # too detailed
        def filter_even_numbers(numbers):
            filtered_numbers = []
            for number in numbers:
                if number % 2 == 0:
                    filtered_numbers.append(number)
            return filtered_numbers


        def square_numbers(numbers):
            squared_numbers = []
            for number in numbers:
                squared_numbers.append(number**2)
            return squared_numbers


        numbers = list(range(20))
        even_numbers = filter_even_numbers(numbers)
        squared_numbers = square_numbers(even_numbers)
        print(squared_numbers)
```

You can even ask for the answer for whole exercises, e.g. “Create a basic framework for a boxplot in Python with matplotlib for two models and 6 instances each that compares the computing time of the models. The time should be one the y-axis and the results from the two models for each instance should be placed next to each other.”

The first result will be this:

```
In [3]: import matplotlib.pyplot as plt
        import numpy as np

        # Example data: Computing times in seconds (replace with actual data)
        model_a_times = [
            [1.2, 1.3, 1.1, 1.4, 1.2],  # Instance 1
            [1.8, 1.7, 1.9, 1.6, 1.8],  # Instance 2
            [2.5, 2.4, 2.6, 2.5, 2.4],  # Instance 3
            [3.1, 3.0, 3.2, 3.1, 3.0],  # Instance 4
```

```
    [1.5, 1.6, 1.4, 1.5, 1.5],  # Instance 5
    [2.0, 2.1, 2.0, 2.2, 2.1],  # Instance 6
]

model_b_times = [
    [1.1, 1.0, 1.2, 1.1, 1.0],  # Instance 1
    [1.6, 1.5, 1.7, 1.6, 1.5],  # Instance 2
    [2.2, 2.3, 2.1, 2.2, 2.3],  # Instance 3
    [2.9, 3.0, 2.8, 2.9, 2.8],  # Instance 4
    [1.4, 1.3, 1.5, 1.4, 1.3],  # Instance 5
    [1.8, 1.7, 1.9, 1.8, 1.7],  # Instance 6
]

# Prepare data for the boxplot
data = []
labels = []

for i in range(6):  # For each instance
    data.append(model_a_times[i])  # Add Model A data
    data.append(model_b_times[i])  # Add Model B data
    labels.append(f"Instance {i+1} - A")  # Label for Model A
    labels.append(f"Instance {i+1} - B")  # Label for Model B

# Boxplot creation
plt.figure(figsize=(12, 6))

# Create the boxplot
box = plt.boxplot(data, patch_artist=True, widths=0.6)

# Customize box colors for the two models
colors = ["lightblue", "lightgreen"] * 6  # Alternating colors for A and B
for patch, color in zip(box["boxes"], colors):
    patch.set_facecolor(color)

# Add title and labels
plt.title("Comparison of Computing Times for Model A and Model B")
plt.ylabel("Computing Time (seconds)")
plt.xlabel("Instances and Models")
plt.xticks(ticks=range(1, len(labels) + 1), labels=labels, rotation=45)

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```



This will probably not be everything you need. Your data may be different, you may have defined different colors or the legend may not yet fit. You should therefore never assume that the first suggestion is already good. Nevertheless, it gives you a basic framework that you can then adapt (again with the AI if necessary).

17.4 A final Note

AI can help you with programming by generating code, detecting errors and explaining complex concepts in a simplified way. It saves time and allows you to focus on the important things. Nevertheless, you should be careful, as AI sometimes provides inefficient or incorrect solutions. Never adopt code you don't understand and always check it critically for safety and functionality. Use AI as support, but work on solving problems yourself in order to be able to program independently in the long term. Programming is learned by thinking and experimenting, not by simply copying.

Chapter 18

Interactive Graphs

18.1 Widgets

The most `bling-bling` of all presentations are interactive ones.

It is the easiest way to get started using IPython's widgets.

Frist we load our modules. And activeta our widget backend via magic command.

```
In [3]: import math  # for some nice functions
        import numpy as np  # for smooth graphs
        import matplotlib.pyplot as plt  # for plotting

        import ipywidgets as widgets  # for interactive widgets

        # to show the animations in the plots we activate the widget backend
        %matplotlib widget
```

18.1.1 The EOQ formula - now interactive

We start with something we know, a nice little function from our first lecture.

The EOQ formula:

$$q^* = \sqrt{\frac{2 \cdot s \cdot \tilde{d}}{h}}$$

minimizes the sum of ordering costs and holding costs.

$$\min(c_{ordering} + c_{holding})$$

s.t.

$$c_{ordering} = \frac{s \cdot \tilde{d}}{q}$$

$$c_{holding} = \frac{h \cdot q}{2}$$

Where:

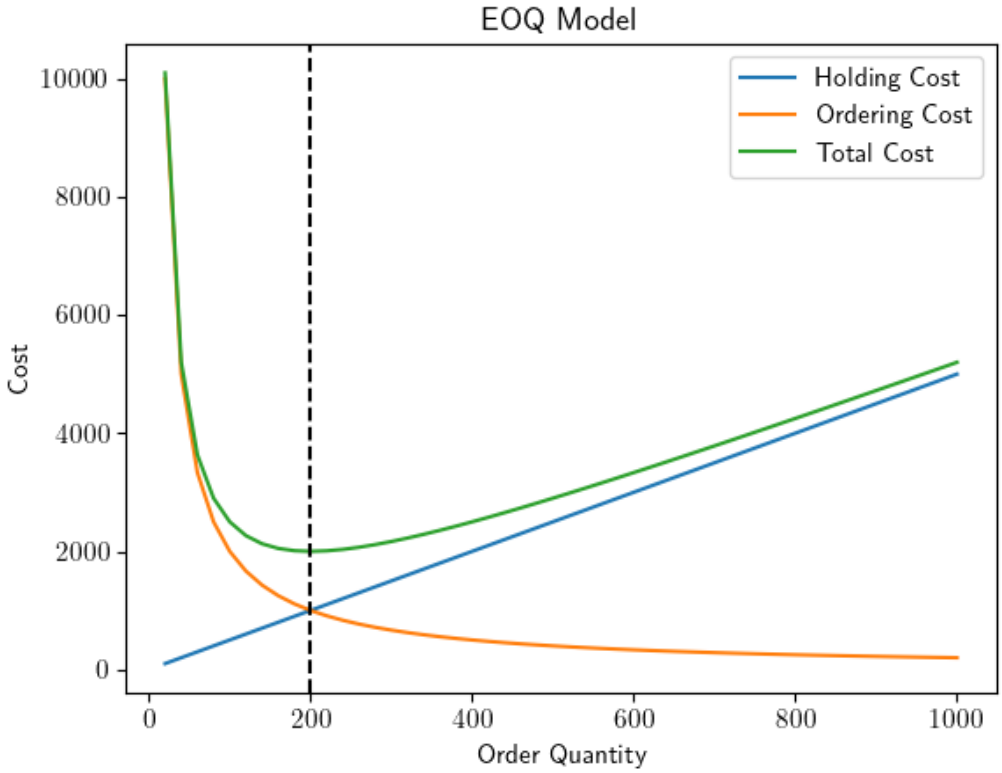
- q^* = Economic Order Quantity
- q = Order quantity
- $\tilde{d}$ = Annual demand (constant)
- s = Ordering cost per order
- h = Holding cost per unit per year

Frist create a function, that plots the EOQ formula with given inputs $\tilde{d}, s, h$ and the axis `ax`, where to visualize the costs.

```
In [ ]: def calculate_EOQ(s: float, d: float, h: float) -> float:
            """
            Calculate Economic Order Quantity (EOQ).

            Parameters:
            - s: Ordering cost per order
            - d: Annual demand (constant)
            - h: Holding cost per unit per year

            Returns:
            - EOQ (Optimal order quantity)
            """
            return math.sqrt((2 * s * d) / h)


        def plot_EOQ(ax, ordering_cost, annual_demand, holding_cost):
            ax.clear()
            eoq_quantity = calculate_EOQ(ordering_cost, annual_demand, holding_cost)

            # to prevent division by zero
            order_quantites = np.linspace(1000, 0, endpoint=False)

            costs_for_ordering = ordering_cost * annual_demand / order_quantites
            costs_for_holding = holding_cost * order_quantites / 2
            total_costs = costs_for_ordering + costs_for_holding

            ax.plot(order_quantites, costs_for_holding, label="Holding Cost")
            ax.plot(order_quantites, costs_for_ordering, label="Ordering Cost")
            ax.plot(order_quantites, total_costs, label="Total Cost")

            ax.axvline(x=eoq_quantity, color="black", linestyle="--")

            ax.legend()
            ax.set_xlabel("Order Quantity")
            ax.set_ylabel("Cost")
            ax.set_title("EOQ Model")
```

we can use this function to plot the graph for a given input.

```
In [32]: ordering_cost = 200  # Replace with your actual ordering cost
         annual_demand = 1000  # Replace with your actual annual demand
         holding_cost = 10  # Replace with your actual holding cost

         fig_fix_eoq, ax_fix_eoq = plt.subplots()

         plot_EOQ(
             ax=ax_fix_eoq,
             ordering_cost=ordering_cost,
             annual_demand=annual_demand,
             holding_cost=holding_cost,
         )
```



See the difference of the matplotlib widget backend and the notebook backend.

18.1.2 Widgets

Now we want to make the graph interactive. We want to change the input values and see the effect on the graph. `widgets.interact` is the function that will help us to do that. Here for more to read: ipywidgets

we take our funktion and pass it to the `interact` function. We also pass the input values as arguments to the `interact` function.

Every input value will be a possible slider in the interactive graph:

- `widgets.IntSlider` creates a slider for integer values
- `widgets.FloatSlider` creates a slider for float values
- `widgets.fixed` creates a fixed value, that can not be changed by the user, so it creates no slider
- `widgets.Checkbox` creates a checkbox for boolean values
- `widgets.Dropdown` creates a dropdown menu for a list of values

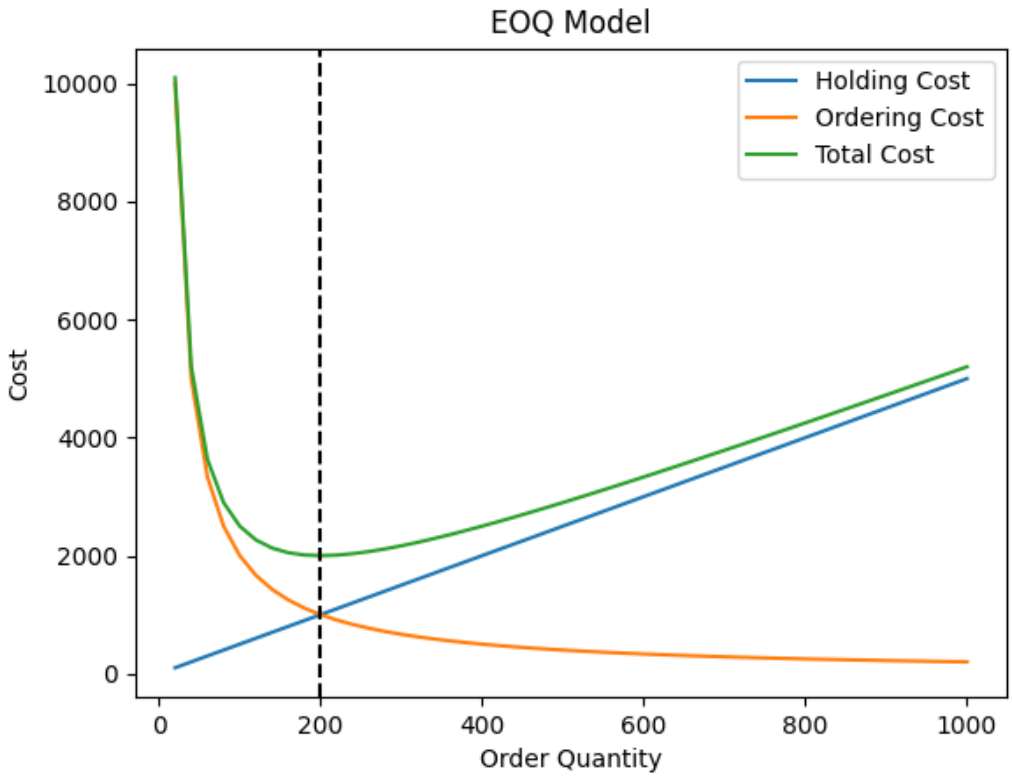
Some things like in which axes is plotted, are not changing, so we must pass them as `widgets.fixed` to the `interact` function.

```
In [9]: fig_interactive_eoq, ax_interactive_eoq = plt.subplots()

        widgets.interact(
            plot_EOQ,
            ax=widgets.fixed(ax_interactive_eoq),
            ordering_cost=widgets.IntSlider(min=0, max=1000, step=10, value=200),
            annual_demand=widgets.IntSlider(min=0, max=2000, step=100, value=1000),
```

```
            holding_cost=widgets.IntSlider(min=0, max=100, step=1, value=10),
        )
```

```
interactive(children=(IntSlider(value=200, description='ordering_cost', max=1000,␣
↪step=10), IntSlider(value=10...
```

```
Out[9]: <function __main__.plot_EOQ(ax, ordering_cost, annual_demand, holding_cost)>
```



Attetntion: interactive graphs and animations can be very challenging for your computer. Consider to close the figures after you are done with them. - You can do that by calling `plt.close()`. Without anything it closes the current figure. - If you want to close a specific figure, you can pass the figure-handler to the function, e.g. `plt.close(fig)`. - Or you can close all figures by calling `plt.close('all')`.

The figures are still displayed, but not interactive anymore.

```
In [10]: plt.close("all")
```

18.2 3D Plots

Matplotlib also supports 3D plots, making them is fairly similar to 2D plots. We just need to set `projection='3d'`.

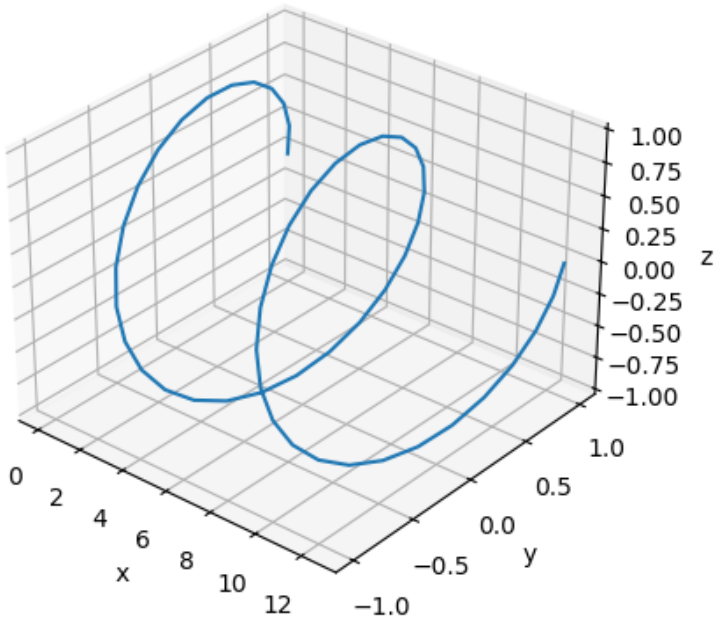
With the widget backend we can play with the graphs interactively.

But we can also set a specific view- and rotation-angle (in degrees) for the 3D plot with `view_init(elev, azim, roll)`.

- `elev` sets the elevation angle in the z plane
- `azim` sets the azimuth angle in the x,y plane
- `roll` sets the roll angle in the x,z plane

We create a 3D plot of the Euler's formula $e^{ix} = \cos(x) + i \cdot \sin(x)$ with i being the imaginary unit.

```
In [13]: # END INTERACTIVE
         # BEGIN STATIC
         # uncomment for interactive mode
         # %matplotlib widget
         # END STATIC

         fig_3d_line, ax_3d_line = plt.subplots(subplot_kw={"projection": "3d"})

         x = np.linspace(0, 4 * np.pi)
         y = np.cos(x)  # same as real part of complex exponential
         z = np.sin(x)  # same as imaginary part of complex exponential

         ax_3d_line.plot(x, y, z)

         ax_3d_line.set_xlabel("x")
         ax_3d_line.set_ylabel("y")
         ax_3d_line.set_zlabel("z")
         ax_3d_line.view_init(
             elev=30, azim=-50, roll=0
         )  # set a good view when working with static images
```



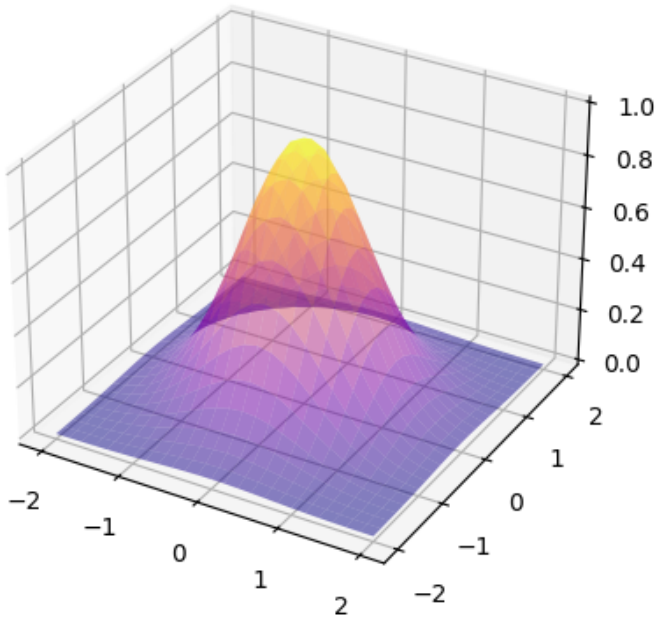
Another common kind of plot that is only possible in 3D is a surface plot: Here we need to generate a grid of x and y values and then calculate the z values for each point in the grid. You can change the look of it via the `cmap` argument. Here you can find a list of possible colormaps: matplotlib colormaps

There are also alternatives to surface plots:

- wireframe
- contour
- trisurf
- scatter3D

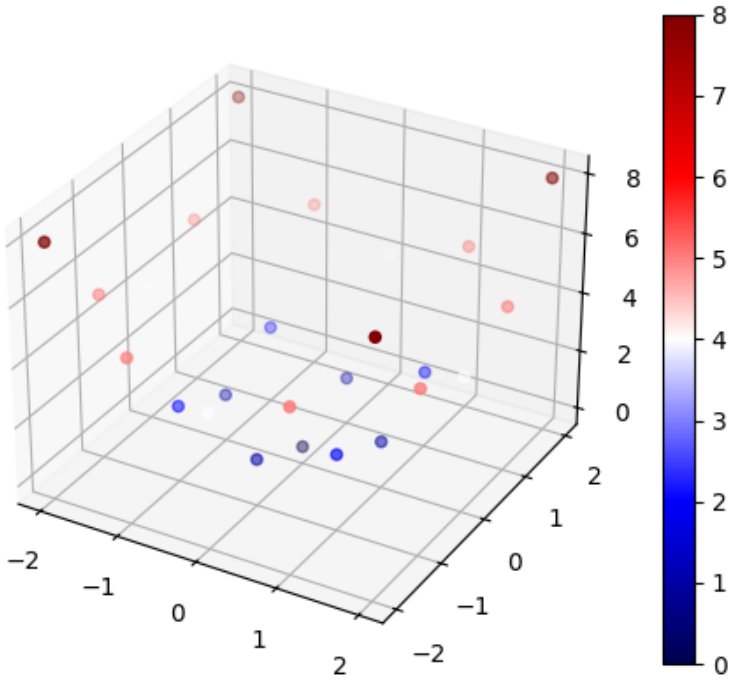
```
In [14]: fig = plt.figure()
         ax = fig.add_subplot(projection="3d")

         ls = np.linspace(-2, 2, 25)
         x, y = np.meshgrid(ls, ls)  # use meshgrid to evaluate a function on a grid
         z = np.exp(-(x**2 + y**2))

         ax.plot_surface(x, y, z, cmap="plasma", alpha=0.5)
         # color and transparency may help
```

```
Out[14]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f7b5952f4d0>
```



```
In [15]: fig = plt.figure()
         ax = fig.add_subplot(projection="3d")

         x_dim = [-2, -1, 0, 1, 2]
         y_dim = [-2, -1, 0, 1, 2]

         x, y = np.meshgrid(x_dim, y_dim)
         # Calculate z values using the function z = exp(-(x^2 + y^2))
         z_list = [(x**2 + y**2) for x, y in zip(x, y)]

         z = np.array(z_list)  # ! convert to numpy array : it must be a 2d-numpy array

         scatter_plot = ax.scatter3D(x, y, z, c=z, cmap="seismic")
         fig.colorbar(scatter_plot)
```

```
Out[15]: <matplotlib.colorbar.Colorbar at 0x7f7b5d3922d0>
```



Things to keep in mind!

While 3D plots can look quite nice and have the potential to visualize things that 2D plots can't, they can quickly become confusing and are often hard to read. Always ask yourself whether you really need a 3D plot.

In most situations, the extra dimension can also be represented using color, size or an extra plot. Your number-one priority should always be readability. If your medium allows for it, use an interactive plot or

an animation so that the 3D plot does not become too confusing. Depending on your type of data, you can also draw lines to a reference plane or line.

18.3 FuncAnimation

Now we want a automatic animation. We can use the `FuncAnimation` function from matplotlib to do that.

```
In [16]: from matplotlib.animation import FuncAnimation
         from IPython.display import Image
```

18.3.1 Draw every frame

To illustrate the concept, we will draw different functions in the same graph and animate them. But first we will visualize the functions without animation.

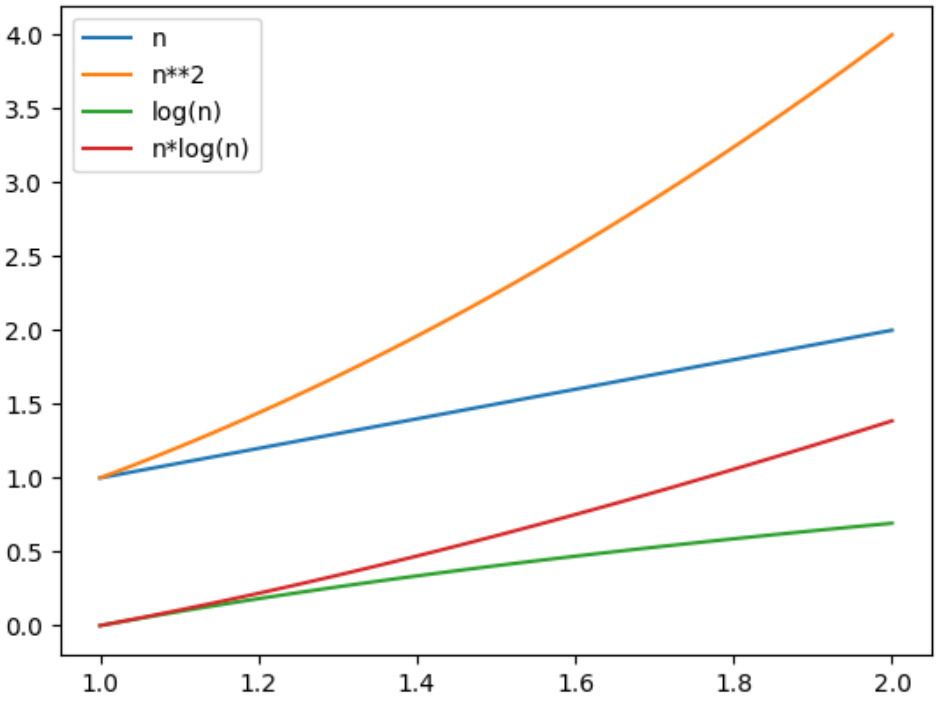
```
In [17]: def O_n_draw_frame(n, ax, show_exp=False):
             ax.clear()
             x = np.linspace(1, n)
             y_linear = x
             y_quadratic = x**2
             y_log = np.log(x)
             y_n_log_n = x * np.log(x)
             y_exp = np.exp(x)
             ax.plot(x, y_linear, label="n")
             ax.plot(x, y_quadratic, label="n**2")
             ax.plot(x, y_log, label="log(n)")
             ax.plot(x, y_n_log_n, label="n*log(n)")
             if show_exp:
                 ax.plot(x, y_exp, label="e^n")
             ax.legend()
```

We can use the widget.interact as before.

```
In [18]: fig_draw_new, ax_draw_new = plt.subplots()
         widgets.interact(
             O_n_draw_frame,
             n=widgets.IntSlider(min=2, max=100, step=1, value=2),
             ax=widgets.fixed(ax_draw_new),
             show_exp=widgets.Checkbox(value=False),
         )
```

```
interactive(children=(IntSlider(value=2, description='n', min=2), Checkbox(value=False,␣
↪description='show_exp'...
```

```
Out[18]: <function __main__.O_n_draw_frame(n, ax, show_exp=False)>
```



or we can use the `FuncAnimation` function from matplotlib. Now we generate a predefined set frames and pass them to the `FuncAnimation` function. These frames will now be visualized as a animation.

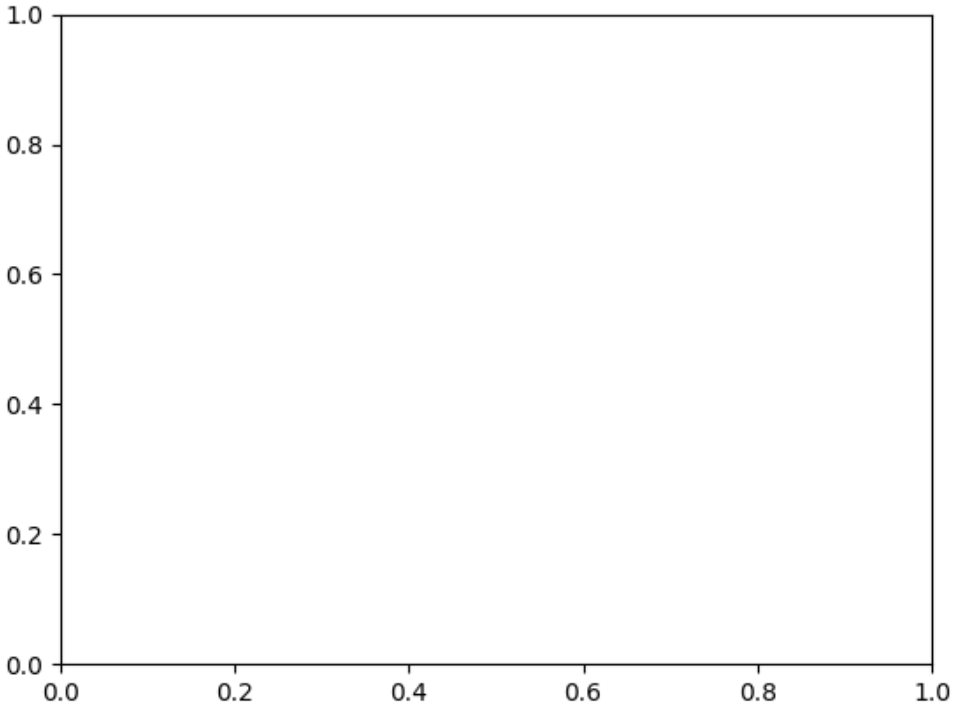
The Synatx is a bit different. instead of passing the `func` to the function and everything else als widgets.Slider or fixed:

`widgets.interact(func, param=widgets.IntSlider(), ax=widgets.fixed(axes))`

we pass the figure and the animation-`func` to the `FuncAnimation` function and everything, thats changing as frames and everything fix as fargs (fargs is everything after the frames parameter in the functioncall, so it is a tuple of all the fixed parameters, the frames must be the first element in the functioncall `func(frames, *fargs)`):

`FuncAnimation(fig, func, frames=range(2, 100), fargs=(axes,))`

```
In [19]: fig_animation, ax_animation = plt.subplots()
         frames = list(range(2, 100))
         anim = FuncAnimation(
             fig_animation, O_n_draw_frame, frames=frames, interval=100,␣
 ↪fargs=(ax_animation,)
         )
```




```
No such comm: 5392401fa89f46a6a42a4aa9a51e775e
```

18.3.2 Update parts of the frame

To This point, we only have made it that way, that every frame is generated, sometimes, we only want to update certain parts of the frame, not the whole axes.

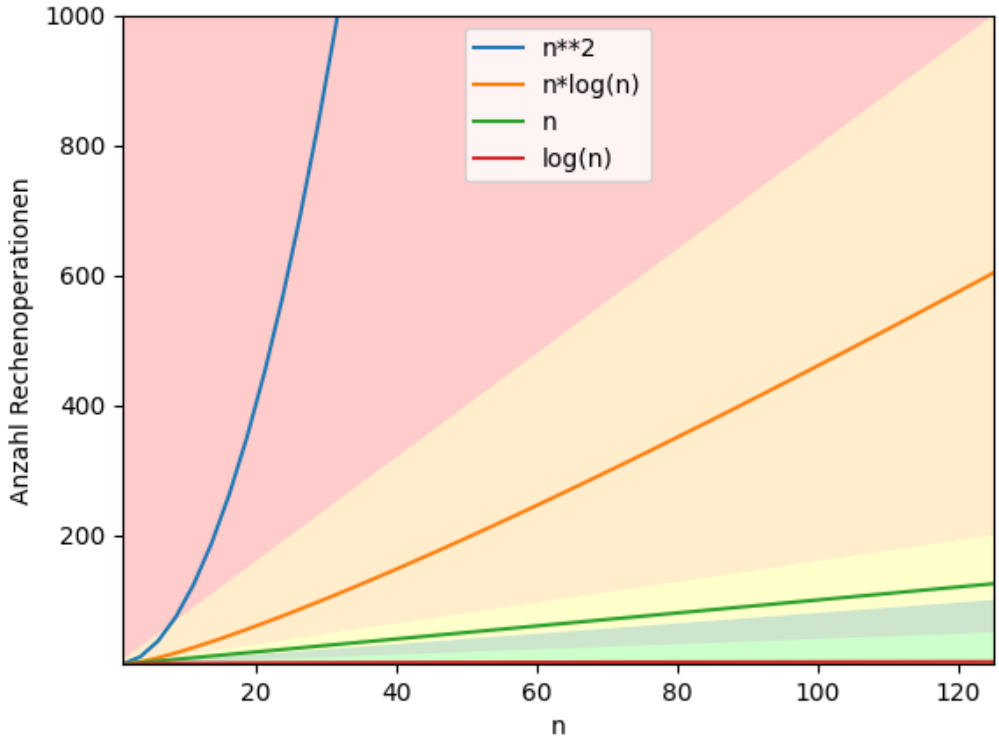
```
In [20]: def O_n_update(n, lines):
             x = np.linspace(1, n)
             for line_name, line in lines.items():
                 line.set_xdata(x)
                 match line_name:
                     case "y_linear":
                         line.set_ydata(x)
                     case "y_quadratic":
                         line.set_ydata(x**2)
                     case "y_log":
                         line.set_ydata(np.log(x))
                     case "y_n_log_n":
                         line.set_ydata(x * np.log(x))
```

We can now outsource our Background and only generate them once.

```
In [21]: def generate_background(ax, x, x_max, y_max):
             sections = [
                 {"start": 0, "color": "lime"},
                 {"start": 0.05, "color": "green"},
                 {"start": 0.1, "color": "yellow"},
                 {"start": 0.2, "color": "orange"},
                 {"start": 1, "color": "red"},
                 {"start": y_max},
             ]

             for section, second_section in zip(sections, sections[1:]):
                 ax.fill_between(
                     x,
                     section["start"] * (y_max / x_max) * x,
                     second_section["start"] * (y_max / x_max) * x,
                     alpha=0.2,
                     facecolor=section["color"],
                     edgecolor="none",
                 )
```

```
In [ ]: fig_update, ax_update = plt.subplots()
        y_max = 1000
        x_max = 1 / 8 * y_max

        x = np.linspace(1, x_max)

        generate_background(ax_update, x, x_max, y_max)

        # we can save our 2D-line objects, to be able to alternate them later
        lines = {
            "y_quadratic": ax_update.plot(x, x**2, label="n**2")[0],
            "y_n_log_n": ax_update.plot(x, x * np.log(x), label="n*log(n)")[0],
            "y_linear": ax_update.plot(x, x, label="n")[0],
            "y_log": ax_update.plot(x, np.log(x), label="log(n)")[0],
        }
        # we can make some settings
        ax_update.legend()
        ax_update.set_xlabel("n")
        ax_update.set_ylabel("Anzahl Rechenoperationen")
        ax_update.set_ylim(1, y_max)
        ax_update.set_xlim(1, x_max)

        if False:
            widgets.interact(
                O_n_update,
                n=widgets.IntSlider(min=2, max=x_max, step=1, value=2),
                lines=widgets.fixed(lines),
            )
        # frames = list(range(2, int(x_max)+1))

        nb_frames = 100
        frames = list(np.linspace(2, x_max, nb_frames))
```

```
        anim = FuncAnimation(fig_update, O_n_update, frames=frames, interval=10,␣
↪fargs=(lines,))
```




```
No such comm: 075c6fa4627748c1bec987d496306eb5
```

```
In [23]: plt.close("all")
```

This can be buggy and its depended on the power of your PC. But we can use this to generate ourselfs a gif and display them instead.

```
In [24]: filename = "../data/own_animation.gif"
         anim.save(filename, fps=30)  # override fps
         # no animations in static version...
         Image(filename)
```

```
MovieWriter ffmpeg unavailable; using Pillow instead.
```

```
Out[24]: <IPython.core.display.Image object>
```

As always, you can also display it directly in markdown.
[title](../data/own_animation.gif ')
Please keep in mind, animations are a nice to have, but in the end, nearly everything can be done with static graphs. And static graphs are easier to understand and to read.

```
In [25]: plt.close("all")
```

18.4 Smooth Translation from 2d to 3d

Using animations, we can also make other changes to the visualization, for example we can change the perspective of the graph.

```
In [26]: def change_angle(angles, ax):
             elev, azim = angles
             ax.view_init(azim, elev)
             if elev > -80:
                 ax.set_yticks([-2, -1, 0, 1, 2])
                 ax.set_ylabel("y")
             else:
                 ax.set_yticks([])
                 ax.set_ylabel("")
```

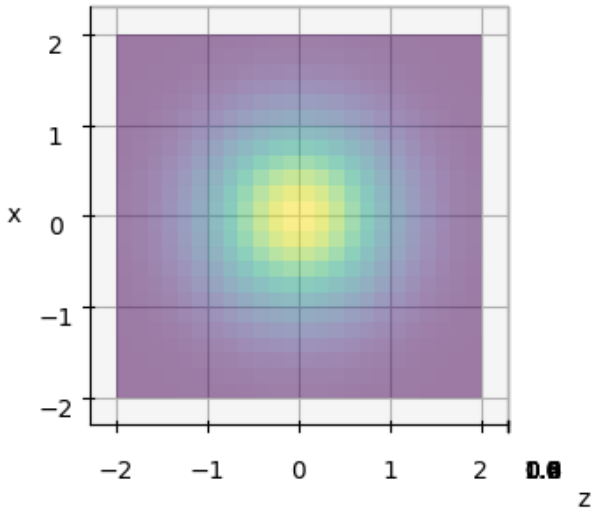
Don't forget to close the figures, if you don't need them anymore. Especially, if you have one big notebook, which is generally not a good Idea.

```
In [27]: plt.close("all")
```

We use the same function as before, but we change the perspective of the graph.
We start at the plan-view XZ, where the y-dimension is invisible. (`elev = -90, azim = 0, roll = 0`)
Then we change the perspective slowly to a 3D view (`elev = -60, azim = 20, roll = 0`).

```
In [28]: fig_transition = plt.figure()
         ax_transition = fig_transition.add_subplot(projection="3d", proj_type="ortho")


         ls = np.linspace(-2, 2, 25)
         x, y = np.meshgrid(ls, ls)  # use meshgrid to evaluate a function on a grid
         z = np.exp(-(x**2 + y**2))

         ax_transition.plot_surface(x, y, z, cmap="viridis", alpha=0.5)
         # color and transparency may help
         ax_transition.set_xlabel("x")
         ax_transition.set_zlabel("z")

         ax_transition.view_init(elev=-90, azim=0)

         nb_frames = 100
         angles_list = [
             (elev, azim)
             for elev, azim in zip(
                 np.linspace(-90, -60, nb_frames), np.linspace(0, 20, nb_frames)
             )
```

```
]

anim_transition = FuncAnimation(
    fig_transition,
    change_angle,
    frames=angles_list,
    interval=10,
    fargs=(ax_transition,),
)
```




```
In [29]: filename = "../data/own_transition.gif"
         anim_transition.save(filename, fps=60)  # override fps
         # no animations in static version...
         Image(filename)
```

```
MovieWriter ffmpeg unavailable; using Pillow instead.
```

```
Out[29]: <IPython.core.display.Image object>
```

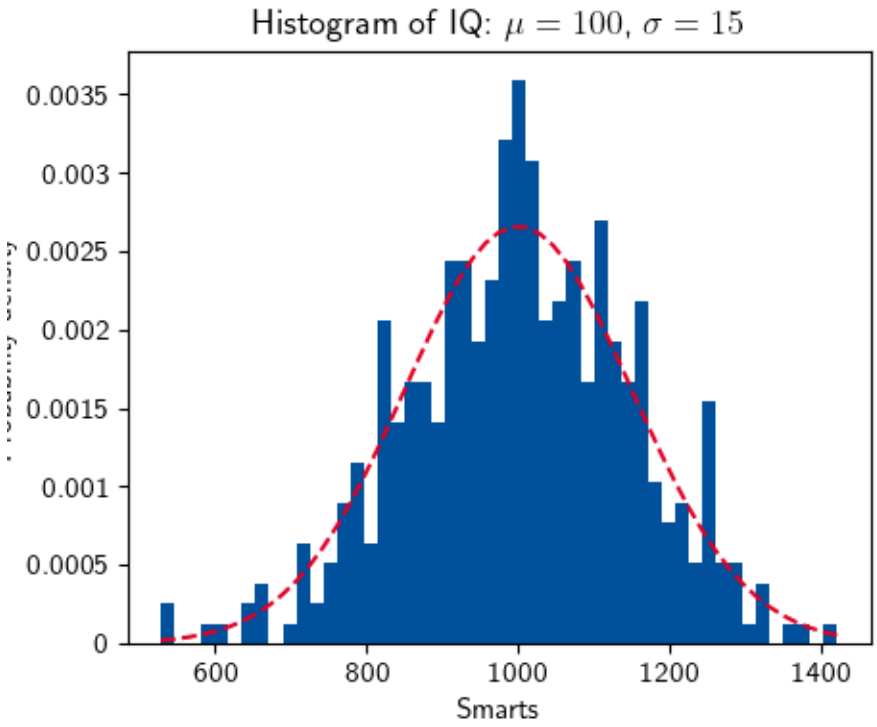
```
In [30]: plt.close("all")
```

18.5 LaTeX

The graphs we have created so far look nice but we want to add them into our presentation, paper or thesis. We could do this by simply saving the graphs as png-files. We already showed you how to do this in lecture 7. Doing this gives us an image which can be a bit blurry, doesn't scale well, the text in it is not searchable and the font is not consistent with the rest of the document. We can avoid all of these problems by saving the graphs as pgf-files. This is a file format that is used by LaTeX to render the text and the graphs. So all those problems we mentioned for png-files are gone. To do this we only have to add some lines of code around the plotting code.

```
In [31]: import matplotlib
         from matplotlib.backends.backend_pgf import FigureCanvasPgf
         import numpy as np
         import matplotlib.pyplot as plt
         from matplotlib.ticker import StrMethodFormatter

         # import locale

         luh_green = (200 / 255, 211 / 255, 23 / 255)
         luh_red = (226 / 255, 0 / 255, 38 / 255)
         luh_blue = (0 / 255, 80 / 255, 155 / 255)
         luh_mid_blue = (153 / 255, 185 / 255, 216 / 255)

         matplotlib.backend_bases.register_backend("pdf", FigureCanvasPgf)
         matplotlib.rcParams.update(
             {
                 "pgf.texsystem": "pdflatex",
                 "font.family": "sans-serif",
                 "font.size": 11,
                 "text.usetex": True,
                 "pgf.rcfonts": False,
                 "font.sans-serif": "Charter",
             }
         )

         np.random.seed(19680801)

         # locale.setlocale(locale.LC_ALL, "de_DE")

         # example data
         mu = 1000  # mean of distribution
         sigma = 150  # standard deviation of distribution
         x = mu + sigma * np.random.randn(437)

         num_bins = 50

         fig_tex, ax_tex = plt.subplots()

         # the histogram of the data
         n, bins, patches = ax_tex.hist(x, num_bins, density=1, color=luh_blue)

         # add a 'best fit' line
         y = (1 / (np.sqrt(2 * np.pi) * sigma)) * np.exp(-0.5 * (1 / sigma * (bins -␣
 ↪mu)) ** 2)
```

```
        ax_tex.plot(bins, y, "--", color=luh_red)
        ax_tex.set_xlabel("Smarts")
        ax_tex.set_ylabel("Probability density")
        ax_tex.set_title(r"Histogram of IQ: $\mu=100$, $\sigma=15$")
        ax_tex.yaxis.set_major_formatter(StrMethodFormatter("{x:n}"))
        ax_tex.xaxis.set_major_formatter(StrMethodFormatter("{x:n}"))

        # Tweak spacing to prevent clipping of ylabel
        # fig.tight_layout()
        fig_tex.set_size_inches(5, 4)
        plt.show()
        plt.savefig("../data/histogram.pgf", format="pgf")
```



To use the image, we only need to include the pgf-file in our LaTeX document.

Chapter 19

Starting with Gurobi - Flower Example

19.1 The Problem

Problem Description

A gardener has a $100m^2$ bed on which he would like to plant sunflowers and/or a maximum of $60m^2$ roses. He has 690 available. He knows the exact cost of planting per m^2 (sunflowers: €6, roses: €9) and the profit (sunflowers: €1, roses: €2).

How many m^2 should he plant with sunflowers and how many with roses to maximize his profit?

The optimal solution x^* is the solution x with which the highest revenue is achieved.

Summarized Data

- budget: 690€
- total bed: 100 m^2
- maximum roses: 60 m^2
- cost: sunflowers: 6€, roses 9€
- profit: sunflowers: 1€, roses 2€

Abstract Mathematical Model

Indices and quantities - $\{\text{roses}, \text{sun}\} \in \mathcal{I}$: flowers

Parameters - e^{roses} and e^{sun}: profit per m^2 per roses and sunflowers - c^{roses} and c^{sun}: cost per m^2 per roses and sunflowers - b: Budget - m^{roses}: maximum area for roses - m^{total}: maximum area for planting

Decision variables - x^{roses} planting area of roses (in m^2)
- x^{sun} planting area of roses (in m^2)

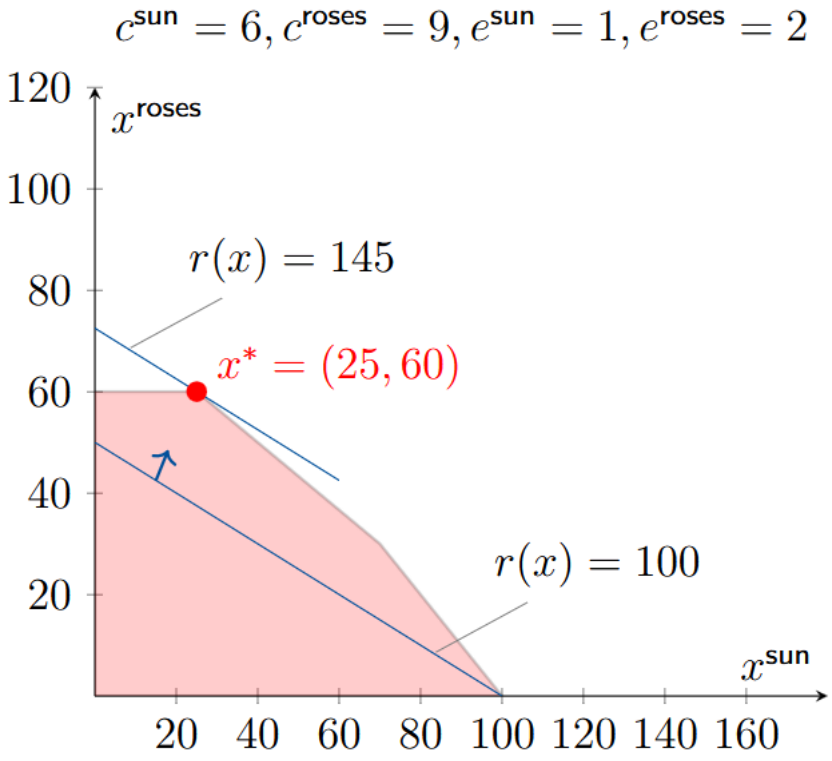
Model (specific for roses and sunflowers)

$$\max \; e^{\text{roses}} \cdot x^{\text{roses}} + e^{\text{sun}} \cdot x^{\text{sun}}$$

$$\text{s. t.}$$

$$\begin{aligned} x^{\text{roses}} + x^{\text{sun}} &\leq m^{\text{total}} \\ c^{\text{roses}} \cdot x^{\text{roses}} + c^{\text{sun}} \cdot x^{\text{sun}} &\leq b \\ x^{\text{roses}} &\leq m^{roses} \end{aligned}$$

The Graphical Solution



19.2 What is Gurobi?

Gurobi is an optimization software for solving mathematical decision problems. It can be used with various programming languages (Python, C++, Julia, R, GAMS).

You might know GAMS from the Operations Management Course in your Bachelor. GAMS is a specialized programming language for implementing mathematical decision problems. Various solvers can be integrated to solve the decision problems, such as Gurobi. The implementation is therefore independent of the solver.

Gurobi can also be used with Python. For this purpose, the package `gurobipy` is used. As the name suggests, the package is specific to the Gurobi solver. To use other solvers, other (solver-dependent and independent) packages are available. If the package `gurobipy` is used, the complete functionality of Gurobi is available.

Installation

Gurobi can be installed via `pip`, see: https://support.gurobi.com/hc/en-us/articles/360044290292-How-do-I-install-Gurobi-for-Python. There are also other ways, but using `pip` works quite well.

If you have installed Gurobi via `pip`, a limited license is included. However, this license is not sufficient for larger models. Therefore, an additional free academic license is required. You can get it at https://www.gurobi.com/features/academic-named-user-license/.

To set up the license on your machine, you need the Gurobi license tools. This is **not** included in the pip installation. Thus, follow this guideline to set up the license: https://support.gurobi.com/hc/en-us/articles/360059842732-How-do-I-set-up-a-license-without-installing-the-full-Gurobi-package.

19.3 Import the Gurobi Library

Make all Gurobi functions and classes available through `gp`.

```
In [1]: import gurobipy as gp
```

As we use `GRB` frequently, we import the class `GRB` explicitly. By that, we can write `GRB.xxx` instead of `gp.GRB.xxx`.

```
In [2]: from gurobipy import GRB
```

19.4 The Data

Create a list `flowers` to store the flowers.

```
In [3]: flowers = ["sunflowers", "roses"]
```

Store the costs and revenues in the dictionarys `costs` and `revenues`.

```
In [4]: costs = {"sunflowers": 6, "roses": 9}
        revenues = {"sunflowers": 1, "roses": 2}
```

Use `max_qm_roses` to set the maximum m^2 for roses, `total_available_qm` to define the available size of the bed and `max_budget` to save the available budget of the gardener.

```
In [5]: max_qm_roses = 60
        total_available_qm = 100
        max_budget = 690
```

19.5 Set up the Gurobi Model

19.5.1 Create a Gurobi model

```
In [6]: flower_model = gp.Model()
```

```
Restricted license - for non-production use only - expires 2026-11-23

```

19.5.2 Add variables to the model

The two variables are added to the model with `addVar()`. Name the variables `X_sunflowers` and `X_roses` and define that the variables are continuous. In addition, set the lower bound `lb` of the variables to 0.

```
In [7]: X_sunflowers = flower_model.addVar(vtype=GRB.CONTINUOUS, name="X_sunflowers",␣
 ↪lb=0)
        X_roses = flower_model.addVar(vtype=GRB.CONTINUOUS, name="X_roses", lb=0)
```

19.5.3 Define the objective function

Define the objective function max $e^{\text{roses}}x^{\text{roses}} + e^{\text{sun}}x^{\text{sun}}$ with the Gurobi function `setObjective` and define the optimization direction.

```
In [8]: flower_model.setObjective(
            (X_sunflowers * revenues["sunflowers"] + X_roses * revenues["roses"]), GRB.
 ↪MAXIMIZE
        )
```

19.5.4 Add Constraints

Define the first constraint $x^{\text{roses}} + x^{\text{sun}} \leq m^{\text{total}}$ with constraint name `max_available_square`.

```
In [9]: flower_model.addConstr(
            (X_sunflowers + X_roses <= total_available_qm), "max_available_square"
        )
```

```
Out[9]: <gurobi.Constr *Awaiting Model Update*>
```

Define the scond constraint $c^{\text{roses}} \cdot x^{\text{roses}} + c^{\text{sun}} \cdot x^{\text{sun}} \leq b$ with constraint name `budget`.

```
In [10]: flower_model.addConstr(
             (costs["sunflowers"] * X_sunflowers + costs["roses"] * X_roses <=␣
 ↪max_budget),
             "budget",
         )
```

```
Out[10]: <gurobi.Constr *Awaiting Model Update*>
```

Define the third constraint $x^{\text{roses}} \leq m^{roses}$ with constraint name `max_roses`.

```
In [11]: flower_model.addConstr((X_roses <= max_qm_roses), "max_roses")
```

```
Out[11]: <gurobi.Constr *Awaiting Model Update*>
```

19.6 Optimize the Gurobi Model

Use `optimize()` to start the Gurobi optimization.

```
In [12]: flower_model.optimize()
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 3 rows, 2 columns and 5 nonzeros
Model fingerprint: 0x22846b2e
Coefficient statistics:
  Matrix range     [1e+00, 9e+00]
  Objective range  [1e+00, 2e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+01, 7e+02]
Presolve removed 1 rows and 0 columns
Presolve time: 0.01s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    2.0000000e+02   5.312500e+01   0.000000e+00      0s
       2    1.4500000e+02   0.000000e+00   0.000000e+00      0s
```

```
Solved in 2 iterations and 0.02 seconds (0.00 work units)
Optimal objective  1.450000000e+02
```

19.7 Analysis

Print the objective function value with `ObjVal`.

```
In [13]: print(f"The objective function value is: {flower_model.ObjVal}")
```

```
The objective function value is: 145.0
```

Print the solution using a loop over the two variables. Use the `X` attribute to get the value of the variable and the `VarName` attribute to get the name of the variable.

```
In [14]: for variable in flower_model.getVars():
             print(f"Variable {variable.VarName} has the value {variable.X}")
```

```
Variable X_sunflowers has the value 25.0
Variable X_roses has the value 60.0
```

Chapter 20

The Classical Transportation Problem

20.1 Mathematical Model Formulation

Problem Description

- At each supply location $i = 1, \ldots, \mathcal{I}$ there are a_i units of a good.
- At each demand location $j = 1, \ldots, \mathcal{J}$ there is a demand for n_j units of the good.
- The sum of the supply quantity equals the sum of the demand quantity.
- The transportation cost for transporting one unit of quantity from supply location i to demand location j is c_{ij} monetary units.

We are looking for a feasible transportation plan with transportation quantities X_{ij} that leads to minimum transportation quantities.

More information: Helber, S., Operations Management Tutorial, S. Helber (2020), 2.Auflage, S. 233 ff.

Abstract Mathematical Model

Indices and quantities - $i \in \mathcal{I}$: facilities - $j \in \mathcal{J}$: customers

Parameters - a_i: quantity of goods available at factory i - c_{ij}: cost of transporting a quantity of goods from facility i to customer j - n_j: quantity of goods demanded by the customer j

Decision variables - $X_{ij} \geq 0$: transport quantity from facility i to customer j

$$\min \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} c_{ij} \cdot X_{ij}$$

s. t.

$$\sum_{j \in \mathcal{J}} X_{ij} = a_i \qquad \forall\, i \in \mathcal{I}$$

$$\sum_{i \in \mathcal{I}} X_{ij} = n_j \qquad \forall\, j \in \mathcal{J}$$

$$X_{ij} \geq 0 \qquad \forall\, i \in \mathcal{I},\ j \in \mathcal{J}$$

Example Data

Table 1: Cost of transportation

facility $\boldsymbol{i}$	customer $\boldsymbol{j}$	1	2	3	4
	1	12	4	12	14
	2	8	18	10	6
	3	16	16	2	12

Table 2: Quantity of goods available

facility i	goods available a_i
1	40
2	50
3	42

Table 3: Quantity of goods demanded

customer j	goods demanded n_j
1	30
2	34
3	44
4	24

20.2 Create the instance

Create a list `factories` (call the factories factory_1, factory_2, ...) and a list `customer_locations` (call the customer_locations customer_1, customer_2, ...). Store the supply quantities of the factories and the demand quantities of the customers in the dictionaries `supply` and `demand`. Create the dictionary `transportation_cost` to store the transportation cost. Use a tuple of factory and customer location as key.

```
In [1]: factories = [f"factory_{nb}" for nb in range(1, 4)]
        customers = [f"customer_{nb}" for nb in range(1, 5)]
```

Table 2: Quantity of goods available

facility i	goods available a_i
1	40
2	50
3	42

Table 3: Quantity of goods demanded

customer j	goods demanded n_j
1	30
2	34
3	44

```
In [2]: # supply and demand
        supply = {"factory_1": 40, "factory_2": 50, "factory_3": 42}
        demand = {"customer_1": 30, "customer_2": 34, "customer_3": 44, "customer_4": 24}
```

Table 1: Cost of transportation

facility i \ customer j	1	2	3	4
1	12	4	12	14
2	8	18	10	6
3	16	16	2	12

```
In [3]: # Parameter: transportation cost
        transportation_cost = {
            ("factory_1", "customer_1"): 12,
            ("factory_1", "customer_2"): 4,
            ("factory_1", "customer_3"): 12,
            ("factory_1", "customer_4"): 14,
            ("factory_2", "customer_1"): 8,
            ("factory_2", "customer_2"): 18,
            ("factory_2", "customer_3"): 10,
            ("factory_2", "customer_4"): 6,
            ("factory_3", "customer_1"): 16,
            ("factory_3", "customer_2"): 16,
            ("factory_3", "customer_3"): 2,
            ("factory_3", "customer_4"): 12,
        }
```

20.3 Optimize with Gurobi

```
In [4]: # the usual Gurobi import
        import gurobipy as gp
        from gurobipy import GRB
```

Set up the empty model `transportation_model`.

```
In [5]: transportation_model = gp.Model()
```

```
Restricted license - for non-production use only - expires 2026-11-23
```

Create all variables at once with `addVars()`. (In comparison to the flower model, where we used `addVar()` to create a single variable.)

```
In [6]: X = transportation_model.addVars(
            factories, customers, vtype=GRB.CONTINUOUS, name="X", lb=0
        )
```

Now we need to create the objective function.

There are several ways for defining the sum in the code. In this case, we use the Gurobi function `quicksum()` (see: https://www.gurobi.com/documentation/9.5/refman/py_quicksum.html).

$$\min \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} c_{ij} \cdot X_{ij}$$

```
In [7]: transportation_model.setObjective(
            (
                gp.quicksum(
                    transportation_cost[factory, customer] * X[factory, customer]
                    for factory in factories
                    for customer in customers
                )
```

```
            ),
            GRB.MINIMIZE,
        )
```

We now start adding the constraints. As for the variables, multiple constraints can be created at once using `addConstrs()`. For the sum, use `quicksum()` again.

Add the first constraint: The entire supply is fully utilized at each factory.

$$\sum_{j \in \mathcal{J}} X_{ij} = a_i \qquad \forall\, i \in \mathcal{I}$$

```
In [8]: transportation_model.addConstrs(
            (
                gp.quicksum(X[factory, customer] for customer in customers) ==␣
 ↪supply[factory]
                for factory in factories
            ),
            "utilize_supply",
        )
```

```
Out[8]: {'factory_1': <gurobi.Constr *Awaiting Model Update*>,
         'factory_2': <gurobi.Constr *Awaiting Model Update*>,
         'factory_3': <gurobi.Constr *Awaiting Model Update*>}
```

The code above works like a list comprehension. Another option is to use a loop. Note that we have to use `addConstr` and not `addConstrs` in this case, as we are adding each constraint separately within the loop.

```
In [9]: for factory in factories:
            transportation_model.addConstr(
                (
                    gp.quicksum(X[factory, customer] for customer in customers)
                    == supply[factory]
                ),
                f"utilize_supply{factory}",
            )
```

Add the second constraint: the entire demand of each customer is satisfied.

$$\sum_{i \in \mathcal{I}} X_{ij} = n_j \qquad \forall\, j \in \mathcal{J}$$

```
In [10]: transportation_model.addConstrs(
             (
                 gp.quicksum(X[factory, customer] for factory in factories) ==␣
 ↪demand[customer]
                 for customer in customers
             ),
             "satisfy_demand",
         )
```

```
Out[10]: {'customer_1': <gurobi.Constr *Awaiting Model Update*>,
          'customer_2': <gurobi.Constr *Awaiting Model Update*>,
```

```
          'customer_3': <gurobi.Constr *Awaiting Model Update*>,
          'customer_4': <gurobi.Constr *Awaiting Model Update*>}
```

Optimize the model using `.optimize()`.

```
In [11]: transportation_model.optimize()
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 10 rows, 12 columns and 36 nonzeros
Model fingerprint: 0x9c77d204
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [2e+00, 2e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [2e+01, 5e+01]
Presolve removed 4 rows and 0 columns
Presolve time: 0.01s
Presolved: 6 rows, 12 columns, 20 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    6.0800000e+02   2.489250e+00   0.000000e+00      0s
       3    6.4400000e+02   0.000000e+00   0.000000e+00      0s

Solved in 3 iterations and 0.02 seconds (0.00 work units)
Optimal objective  6.440000000e+02
```

20.4 Analysis

Print the total transportation costs and the transportation quantities between all factories and customers greater than 0.

```
In [12]: # objective function value
         print(f"The total transportation costs are {transportation_model.ObjVal}.")
```

```
The total transportation costs are 644.0.
```

```
In [13]: # transportation quantities
         for factory in factories:
             for customer in customers:
                 if X[factory, customer].X > 0:
                     print(
                         f"We transport {X[factory, customer].X} units from {factory} to␣
 ↪{customer}."
                     )
```

```
We transport 4.0 units from factory_1 to customer_1.
We transport 34.0 units from factory_1 to customer_2.
We transport 2.0 units from factory_1 to customer_3.
We transport 26.0 units from factory_2 to customer_1.
We transport 24.0 units from factory_2 to customer_4.
We transport 42.0 units from factory_3 to customer_3.
```

Chapter 21

Gurobi - Parameters, Attributes and Environments

21.1 Defining the models

We use the simple model from the exercise and the mip_1 example from Gurobi as examples to show the functionality of an parameters, attributes and environments.

```
In [1]: # import statements
        import gurobipy as gp
        from gurobipy import GRB
        import os
```

Start with defining the first model. As with other things in Python, we can define the model within a function. To create the model, we can later just call the function.

First model:

$$\begin{aligned} &\min\ 6X_1 + 5X_2 + 3Y \\ &\text{s. t.} \\ &\quad X_1 + X_2 \geq 6 \\ &\quad 2X_1 + 3X_2 - 4Y \leq 0 \\ &\quad X_1, X_2 \geq 0 \\ &\quad Y \in \{0, 1, \ldots, 7\} \end{aligned}$$

```
In [2]: # if we want to use the function without passing the environment, we can use a␣
        ↪default value of my_env=None
        def define_model_1(my_env=None):
            simple_model_1 = gp.Model(env=my_env)

            # Decision variables
            X1 = simple_model_1.addVar(vtype=GRB.CONTINUOUS, name="X1", lb=0)
            X2 = simple_model_1.addVar(vtype=GRB.CONTINUOUS, name="X2", lb=0)
            Y = simple_model_1.addVar(vtype=GRB.INTEGER, name="Y", lb=0, ub=7)

            # Objective function
            simple_model_1.setObjective((6 * X1 + 5 * X2 + 3 * Y), GRB.MINIMIZE)

            # Constraints
            # First constraint
```

```
        simple_model_1.addConstr((X1 + X2 >= 6), "first")

        # Second constraint
        simple_model_1.addConstr((2 * X1 + 3 * X2 - 4 * Y <= 0), "second")

        return simple_model_1, Y
```

Define also the second model within a function.
Second model:

$$\max x + y + 2z$$

$$\begin{aligned} \text{s. t.} \\ x + 2y + 3z &\leq 4 \\ x + y &\geq 1 \\ x, y, z &\in \{0, 1\} \end{aligned}$$

```
In [3]: # if we always want to pass the environment to the model, we can define the␣
 ↪model as follows
        def define_model_2(my_env):

            simple_model_2 = gp.Model(env=my_env)

            X = simple_model_2.addVar(vtype=GRB.BINARY, name="X")
            Y = simple_model_2.addVar(vtype=GRB.BINARY, name="Y")
            Z = simple_model_2.addVar(vtype=GRB.BINARY, name="Z")

            # Set objective
            simple_model_2.setObjective(X + Y + 2 * Z, GRB.MAXIMIZE)

            # Add constraint: x + 2 y + 3 z <= 4
            simple_model_2.addConstr(X + 2 * Y + 3 * Z <= 4, "first")

            # Add constraint: x + y >= 1
            simple_model_2.addConstr(X + Y >= 1, "second")

            # return variable Z, as we need the variable later on
            return simple_model_2
```

21.2 Parameters

Let's create the first model by calling the function `define_model_1()`.

```
In [4]: simple_model_1, Y = define_model_1()
        simple_model_1.optimize()
```

```
Restricted license - for non-production use only - expires 2026-11-23
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
```

```
Optimize a model with 2 rows, 3 columns and 5 nonzeros
Model fingerprint: 0x1f88faca
Variable types: 2 continuous, 1 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 4e+00]
  Objective range  [3e+00, 6e+00]
  Bounds range     [7e+00, 7e+00]
  RHS range        [6e+00, 6e+00]
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 5 nonzeros
Variable types: 2 continuous, 1 integer (0 binary)
Found heuristic solution: objective 51.0000000

Root relaxation: objective 4.350000e+01, 2 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0   43.50000    0    1   51.00000   43.50000  14.7%     -    0s
H    0     0                       45.0000000   43.50000  3.33%     -    0s
*    0     0               0       44.0000000   44.00000  0.00%     -    0s

Explored 1 nodes (4 simplex iterations) in 0.06 seconds (0.00 work units)
Thread count was 8 (of 8 available processors)

Solution count 3: 44 45 51

Optimal solution found (tolerance 1.00e-04)
Best objective 4.400000000000e+01, best bound 4.400000000000e+01, gap 0.0000%
```

The parameters control how the Gurobi solver works. All parameters have a default value. You can change them before optimization begins.

Let's define a solution limit. During optimization, Gurobi usually finds several solutions that are constantly improved. With a solution limit, Gurobi will terminate after the defined number of solutions found. In this example, we set the solution limit to 1 meaning Gurobi will terminate after the first solution found. We need to use `.reset()` to discard the solution above.

```
In [5]: simple_model_1.reset()
        simple_model_1.Params.SolutionLimit = 1
        simple_model_1.optimize()
```

```
Discarded solution information
Set parameter SolutionLimit to value 1
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
SolutionLimit  1

Optimize a model with 2 rows, 3 columns and 5 nonzeros
Model fingerprint: 0x1f88faca
```

```
Variable types: 2 continuous, 1 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 4e+00]
  Objective range  [3e+00, 6e+00]
  Bounds range     [7e+00, 7e+00]
  RHS range        [6e+00, 6e+00]
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 5 nonzeros
Variable types: 2 continuous, 1 integer (0 binary)
Found heuristic solution: objective 51.0000000

Explored 0 nodes (0 simplex iterations) in 0.03 seconds (0.00 work units)
Thread count was 8 (of 8 available processors)

Solution count 1: 51

Solution limit reached
Best objective 5.100000000000e+01, best bound 9.000000000000e+00, gap 82.3529%
```

We can reset all parameters to the default value with `resetParams()`

```
In [6]: simple_model_1.resetParams()
```

```
Reset all parameters
```

We can turn off logging to consol with `LogToConsole = 0`.

```
In [7]: simple_model_1.Params.LogToConsole = 0
        simple_model_1.optimize()
        print(simple_model_1.ObjVal)
```

```
Set parameter LogToConsole to value 0
44.0
```

There are alternative ways (styles) for setting parameters. They all lead to the same result.

```
In [8]: # simple_model_1.setParam('LogToConsole', 0)
        # simple_model_1.setParam(GRB.Param.LogToConsole, 0)

        # the case of the parameter name is ignored
        # simple_model_1.Params.logtoconsole = 0
```

Other examples of parameters that you will often need are the **time limit** and the **mip gap**. More about parameters can be found in the parameter section of the Gurobi homepage.

21.3 Attributes

A Gurobi model contains a lot of information. Most of it is stored as attributes. There are different groups of attributes. Some refer to the model itself, others to the variables or the constraints.

You already know some of the attributes. E.g. the attribute `.ObjVal` will give you the objective function value.

```
In [9]: print(f"OFV of simple_model_1: {simple_model_1.ObjVal}")

        # there are also different ways to access attributes
        # simple_model_1.getAttr(GRB.Attr.ObjVal)
```

```
OFV of simple_model_1: 44.0
```

`.X`, which gives us the value of a variable after optimization, is also an attribute.

```
In [10]: print(f"Y: {Y.X}")
         print(f"X1: {simple_model_1.getVarByName('X1').X}")
```

```
Y: 4.0
X1: 2.0
```

Another example is the number of variables in the model which can be accessed with `NumVars`.

```
In [11]: print(f"Number of variables: {simple_model_1.NumVars}")
```

```
Number of variables: 3
```

All those attributes above can be accessed but not changed by the user. However, there are also attributes that can be changed, e.g. the name of the constraints, the variable names or the bounds of a variable.

Let's change the lower bound for our variable `Y` to 5.

```
In [12]: print(f"Y: {Y.X}")
         simple_model_1.reset()  # reset the model
         Y.lb = 5
         simple_model_1.optimize()
         print(f"Y: {Y.X}")
```

```
Y: 4.0
Y: 5.0
```

More about attributes can be found in the attribute section of the Gurobi homepage.

21.4 Using environments

A standard environment is created automatically in `gurobipy`. However, we can also create an environment ourselves. This can be particularly useful when solving multiple models sequentially or iteratively.

For an environment, we can define **global parameters** that apply to all models within the environment. In addition, we can set **local parameters** for each model as before.

You can find out more about environments on the Gurobi homepage.

Create an empty Gurobi environment.

```
In [13]: my_env = gp.Env()
```

```
Restricted license - for non-production use only - expires 2026-11-23
```

We can specify settings for the entire environment that apply to all models within the environment. E.g. we can use the parameter `LogToConsole` to turn off logging in the consol.

```
In [14]: my_env.setParam("LogToConsole", 0)
```

```
Set parameter LogToConsole to value 0
```

We can now create a the `simple_model_1` and the `simple_model_2` within this environment. Therefore, we need to call the functions for defining the models and pass the environment `my_env` as an argument.

```
In [15]: simple_model_1, Y = define_model_1(my_env)
         simple_model_2 = define_model_2(my_env)
```

The models can be optimized as usual.

```
In [16]: simple_model_1.optimize()
         print(f"OFV of simple_model_1: {simple_model_1.ObjVal}")
```

```
OFV of simple_model_1: 44.0
```

```
In [17]: simple_model_2.optimize()
         print(f"OFV of simple_model_2: {simple_model_2.ObjVal}")
```

```
OFV of simple_model_2: 3.0
```

Let's assume you want to store the logging information of the `simple_model_1` in a textfile (but not from the first model). You can do this by simply setting the local parameter for this model.

```
In [18]: result_directory = "../results/lecture10"
         if not os.path.exists(result_directory):
             os.makedirs(result_directory)

         simple_model_1.Params.LogFile = f"{result_directory}/Results_simple_model_1.txt"
```

```
In [19]: simple_model_1.reset()
         simple_model_2.reset()

         simple_model_1.optimize()
         simple_model_2.optimize()
```

At the end, we need to close the models and the environment.

```
In [20]: simple_model_1.dispose()
         simple_model_2.dispose()
         my_env.dispose()
```

Note: You can also use the `with open()` syntax for environments (as for text files). This will automatically close the environment for you. However, your code will be one block intended.

When should environments be used?

When you just solve one model, it is absolutely sufficient to use the default environment that is created by gurobipy. We will do this in following in most cases.

If you work with multiple models, use an environment to set up global parameters only once. In addition, when working with remote resources, you will only need one license instead of one for each model.

Chapter 22

Capacitated Vehicle Routing Problem

22.1 Mathematical Model Formulation

Indices and Sets

- $i, j \in \mathcal{I} = \{1, \ldots, I\}$: locations with location 1 as depot
- $m \in \mathcal{M} = \{1, \ldots, M\}$: vehicle tours

Parameters

- c_{ij}: Distance between location i and j
- w_i: Transport units at location i
- b: vehicle capacity

Decision Variables

- $X_{ijm} \in \{0, 1\}$: 1, if the vehicle drives in tour m from location i to location j and 0 else
- $Y_{im} \in \{0, 1\}$: 1, if location i is included in tour m and 0 else
- $Z_i \in \mathbb{R}$: reel-value auxiliary variable to avoid short cycles

$$\min \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} \sum_{m \in \mathcal{M}} c_{ij} \cdot X_{ijm}$$

s. t.

$$\sum_{i \in \mathcal{I}} w_i \cdot Y_{im} \leq b \qquad \forall\, m \in \mathcal{M}$$

$$\sum_{j \in \mathcal{J}} X_{ijm} = Y_{im} \qquad \forall\, i \in \mathcal{I},\ m \in \mathcal{M}$$

$$\sum_{i \in \mathcal{I}} X_{ijm} = Y_{im} \qquad \forall\, j \in \mathcal{J},\ m \in \mathcal{M}$$

$$\sum_{m \in \mathcal{M}} Y_{im} = 1 \qquad \forall\, i = 2, \ldots, I$$

$$Z_i - Z_j + I \cdot \sum_{m \in \mathcal{M}} X_{ijm} \leq I - 1 \quad \forall\, i, j = 2, \ldots, I, i \neq j$$

$$X_{iim} = 0 \qquad \forall\, i \in \mathcal{I},\ m \in \mathcal{M}$$

—

```
In [39]: import pickle
         import gurobipy as gp
         from gurobipy import GRB
```

```
In [40]: import sys

         sys.path.append("../data/lecture_08")

         import instance_module_vrp
```

22.2 Data

Before we start using our generated (random) instances, we will use the data from Helber (2020, p. 243) as an example. It is **ALWAYS** a good idea to start your first steps with a model with a **very small instance**, even if the instance is not realistic. If you have an instance where you know the solution, you should use this instance. If this is not the case, use an instance where you can compute the solution by hand or where you can comprehend the found solution easily.

```
In [41]: # The data from Helber (2020, p. 243) is given in the following:
         locations = [f"l{l}" for l in range(1, 12)]
         distance_list = [
             ("l1", "l2", 9),
             ("l1", "l3", 5),
             ("l1", "l4", 5),
             ("l1", "l5", 9),
             ("l1", "l6", 7),
             ("l1", "l7", 7),
             ("l1", "l8", 4),
             ("l1", "l9", 6),
             ("l1", "l10", 11),
             ("l1", "l11", 11),
             ("l2", "l3", 4),
             ("l2", "l4", 10),
             ("l2", "l5", 14),
             ("l2", "l6", 14),
             ("l2", "l7", 16),
             ("l2", "l8", 13),
             ("l2", "l9", 7),
             ("l2", "l11", 10),
             ("l2", "l10", 6),
             ("l3", "l4", 6),
             ("l3", "l5", 10),
             ("l3", "l6", 10),
             ("l3", "l7", 12),
             ("l3", "l8", 9),
             ("l3", "l9", 5),
             ("l3", "l10", 10),
             ("l3", "l11", 10),
             ("l4", "l5", 4),
             ("l4", "l6", 6),
             ("l4", "l7", 8),
             ("l4", "l8", 7),
```

```
            ("l4", "l9", 11),
            ("l4", "l10", 16),
            ("l4", "l11", 16),
            ("l5", "l6", 4),
            ("l5", "l7", 6),
            ("l5", "l8", 11),
            ("l5", "l9", 15),
            ("l5", "l10", 20),
            ("l5", "l11", 20),
            ("l6", "l7", 4),
            ("l6", "l8", 9),
            ("l6", "l9", 13),
            ("l6", "l10", 18),
            ("l6", "l11", 18),
            ("l7", "l8", 5),
            ("l7", "l9", 9),
            ("l7", "l10", 14),
            ("l7", "l11", 14),
            ("l8", "l9", 6),
            ("l8", "l10", 9),
            ("l8", "l11", 9),
            ("l9", "l10", 5),
            ("l9", "l11", 5),
            ("l10", "l11", 4),
        ]

        distance = {}
        for l1, l2, d in distance_list:
            distance[(l1, l2)] = d
            distance[(l2, l1)] = d
        for l in locations:
            distance[(l, l)] = 0

        transport_units = {
            "l1": 0,
            "l2": 5,
            "l3": 2,
            "l4": 1,
            "l5": 4,
            "l6": 5,
            "l7": 4,
            "l8": 3,
            "l9": 2,
            "l10": 3,
            "l11": 4,
        }
        instance_number = 1
        capacity = 5
```

With this data create a instance from the CVRP class.

```
In [42]: instance = instance_module_vrp.VrpData(
             instance_number=instance_number,
             capacity=capacity,
```

```
            locations=locations,
            distance=distance,
            transport_units=transport_units,
        )
```

22.3 Build the Model

In this example, we use the default environment created by gurobipy automatically again. Set up an empty model.

```
In [43]: vrp = gp.Model()
```

Now create the variables and constraints of the model using a function.

```
In [44]: def build_vrp(vrp, instance):
             # Variables
             X = vrp.addVars(
                 instance.locations,
                 instance.locations,
                 instance.tours,
                 vtype=GRB.BINARY,
                 name="X",
             )
             Y = vrp.addVars(instance.locations, instance.tours, vtype=GRB.BINARY,␣
 ↪name="Y")
             Z = vrp.addVars(instance.locations, vtype=GRB.CONTINUOUS, name="Z")

             # define the objective function
             vrp.setObjective(
                 (
                     gp.quicksum(
                         instance.distance[(i, j)] * X[i, j, m]
                         for i in instance.locations
                         for j in instance.locations
                         for m in instance.tours
                     )
                 ),
                 GRB.MINIMIZE,
             )

             # constraints
             # First constraint: capacity restriction of the tours
             vrp.addConstrs(
                 (
                     gp.quicksum(
                         instance.transport_units[i] * Y[i, m] for i in instance.
 ↪locations
                     )
                     <= instance.capacity
                     for m in instance.tours
                 ),
                 "capacity",
             )
```

```
            # Second constraint: if a location is included in the tour m, this location␣
↪must be left exactly once.
            vrp.addConstrs(
                (
                    gp.quicksum(X[i, j, m] for j in instance.locations) == Y[i, m]
                    for i in instance.locations
                    for m in instance.tours
                ),
                "location_must_be_left",
            )

            # Third constraint: if a location is included in the tour m, this location␣
↪must be approached exactly once.
            vrp.addConstrs(
                (
                    gp.quicksum(X[i, j, m] for i in instance.locations) == Y[j, m]
                    for j in instance.locations
                    for m in instance.tours
                ),
                "location_approached",
            )

            # Fourth constraint: each location must be assigned to exactly one tour.
            vrp.addConstrs(
                (
                    gp.quicksum(Y[i, m] for m in instance.tours) == 1
                    for i in instance.locations[1:]
                ),
                "one_assignment",
            )

            # Fifth constraint: elimination of short cycles without the depot.
            vrp.addConstrs(
                (
                    Z[i]
                    - Z[j]
                    + len(instance.locations) * gp.quicksum(X[i, j, m] for m in␣
↪instance.tours)
                    <= len(instance.locations) - 1
                    for i in instance.locations[1:]
                    for j in instance.locations[1:]
                    if i != j
                ),
                "short_cycles",
            )

            # Sixth constraint: no location can be approached from itself.
            vrp.addConstrs(
                (X[i, i, m] == 0 for i in instance.locations for m in instance.tours),␣
↪"self"
            )

            # we can return the variables, as we will need them later on for analysis
```

```
            return X, Y
```

Now we call the function to add the variables and constraints to our model.

```
In [45]: X, Y = build_vrp(vrp, instance)
```

We can now write the LP file to analyze the created model with the instance data.

```
In [46]: import os

         path = "../results/lecture_11"
         if not os.path.exists(path):
             os.makedirs(path)
         vrp.write(f"{path}/vrp.lp")
```

22.4 Optimize Model

Optimize the model.

```
In [47]: vrp.optimize()
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0xbde8918a
Variable types: 11 continuous, 1320 integer (1320 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]
Presolve removed 180 rows and 841 columns
Presolve time: 0.10s
Presolved: 260 rows, 490 columns, 1640 nonzeros
Variable types: 10 continuous, 480 integer (480 binary)
Found heuristic solution: objective 134.0000000

Root relaxation: objective 1.050909e+02, 275 iterations, 0.02 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  105.09091    0   21  134.00000  105.09091  21.6%     -    0s
H    0     0                       126.0000000  105.09091  16.6%     -    0s
     0     0  110.00000    0   37  126.00000  110.00000  12.7%     -    0s
     0     0  110.00000    0   24  126.00000  110.00000  12.7%     -    0s
     0     0  110.00000    0   21  126.00000  110.00000  12.7%     -    0s
     0     0  115.00000    0   41  126.00000  115.00000  8.73%     -    0s
     0     0  115.00000    0   38  126.00000  115.00000  8.73%     -    0s
```

```
     0     0  118.00000    0   41  126.00000  118.00000  6.35%     -    0s
     0     0  118.00000    0   33  126.00000  118.00000  6.35%     -    0s
     0     0  118.00000    0   48  126.00000  118.00000  6.35%     -    0s
     0     0  118.00000    0   26  126.00000  118.00000  6.35%     -    0s
     0     2  122.00000    0   24  126.00000  122.00000  3.17%     -    0s

Cutting planes:
  Learned: 5
  Gomory: 9
  Cover: 1
  Implied bound: 1
  Clique: 52
  MIR: 3
  Zero half: 1

Explored 11 nodes (1047 simplex iterations) in 0.49 seconds (0.07 work units)
Thread count was 8 (of 8 available processors)

Solution count 2: 126 134

Optimal solution found (tolerance 1.00e-04)
Best objective 1.260000000000e+02, best bound 1.260000000000e+02, gap 0.0000%
```

22.5 Model Status Code

Depending on the current stage and the result of the model, the model has a different status code. The status code can be accessed with the attribute `Status`. A list of all status codes is given at the homepage. We now want to print the status code of our model.

```
In [48]: status_code = vrp.Status
         print(f"The status code is {status_code}")
```

```
The status code is 2
```

```
In [49]: # We can also check if the model status is equal to a certain value
         if vrp.Status == GRB.OPTIMAL:
             print('Yes, the model has the status "optimal".')
         else:
             print('No, the status code is not "optimal".')
```

```
Yes, the model has the status "optimal".
```

Why is it useful to check the status code after optimization?

Let's consider the following example: We assume that the second location cannot be reached by any vehicle tour. We therefore set the upper bound for the variable Y_{im} for $i = \text{l2}$ and all $m \in \mathcal{M}$ to 0, so we are not able to include location l2 in any tour.

```
In [50]: for m in instance.tours:
             Y["l2", m].ub = 0
```

```
In [51]: vrp.optimize()
         status_code = vrp.Status
         print(f"The status code is {status_code}")
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0xf4060a0f
Variable types: 11 continuous, 1320 integer (1320 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]

MIP start from previous solve did not produce a new incumbent solution

Presolve removed 0 rows and 11 columns
Presolve time: 0.00s

Explored 0 nodes (0 simplex iterations) in 0.03 seconds (0.00 work units)
Thread count was 1 (of 8 available processors)

Solution count 0

Model is infeasible
Best objective -, best bound -, gap -
The status code is 3
```

If we now want to retrieve the OFV as usual, we get an error as the model has no OFV.

```
In [52]: # print(f"The OFV of the model is {vrp.ObjVal}.")  # not working
```

Thus, it is a good idea to check the status code before analyzing the model. Most commonly the status code of a model you analyze will be optimal, infeasible, or time limit. The status code time limit indicates that the model was not solved to optimality within the time limit you set. This could go in both directions. Either a feasible solution could not yet be proven to optimality or the infeasibility of the model is not yet proven. In the first case we still want to get the best solution found so far, but because of the second case we cannot just check the status and retrieve the OFV. We have to check if the solver found a feasible solution first and then retrieve the OFV.

```
In [53]: if status_code == GRB.OPTIMAL:
             print(f"The OFV of the model is {vrp.ObjVal}.")
         elif status_code == GRB.TIME_LIMIT:
             if vrp.SolCount > 0:
                 print(f"The model has a feasible solution, but the time limit was␣
 ↪reached.")
             else:
```

```
            print(f"No feasible solution was found within the time limit.")
        elif status_code == GRB.INFEASIBLE:
            print("The model is infeasible, we cannot retrieve a OFV.")
        else:
            print(f"The model has another status code (code {status_code}).")
```

```
The model is infeasible, we cannot retrieve a OFV.
```

In [54]:
```
# re-set the upper bound to 1 and solve the model again
for m in instance.tours:
    Y["12", m].ub = 1
vrp.optimize()
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0xbde8918a
Variable types: 11 continuous, 1320 integer (1320 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]

MIP start from previous solve produced solution with objective 126 (0.03s)
Loaded MIP start from previous solve with objective 126

Presolve removed 180 rows and 841 columns
Presolve time: 0.07s
Presolved: 260 rows, 490 columns, 1640 nonzeros
Variable types: 10 continuous, 480 integer (480 binary)

Root relaxation: objective 1.050909e+02, 275 iterations, 0.01 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  105.09091    0   21  126.00000  105.09091  16.6%     -    0s
     0     0  110.00000    0   31  126.00000  110.00000  12.7%     -    0s
     0     0  110.00000    0   28  126.00000  110.00000  12.7%     -    0s
     0     0  110.00000    0   21  126.00000  110.00000  12.7%     -    0s
     0     0  115.00000    0   30  126.00000  115.00000  8.73%     -    0s
     0     0  118.00000    0   34  126.00000  118.00000  6.35%     -    0s
     0     0  118.00000    0   28  126.00000  118.00000  6.35%     -    0s
     0     0  118.00000    0   33  126.00000  118.00000  6.35%     -    0s
     0     0  118.00000    0   32  126.00000  118.00000  6.35%     -    0s
     0     0  118.00000    0   32  126.00000  118.00000  6.35%     -    0s
     0     2  122.00000    0   26  126.00000  122.00000  3.17%     -    0s
```

```
Cutting planes:
  Learned: 5
  Gomory: 7
  Cover: 3
  Implied bound: 2
  Clique: 42
  MIR: 9
  Mod-K: 4

Explored 16 nodes (1072 simplex iterations) in 0.56 seconds (0.07 work units)
Thread count was 8 (of 8 available processors)

Solution count 1: 126

Optimal solution found (tolerance 1.00e-04)
Best objective 1.260000000000e+02, best bound 1.260000000000e+02, gap 0.0000%
```

22.6 Analyze the model and the solution structure

Let's save the objective function value in the variable `ofv_1` and print the variable.

```
In [55]: ofv_1 = vrp.ObjVal
         print(ofv_1)
```

```
126.0
```

Let's solve the model again with a MIP Gap of 20%. Remember to reset the model to discard any previously computed solution information. Check how to set the MIP Gap on the Gurobi documentation: https://www.gurobi.com/documentation/current/refman/parameters.html

Keep in mind that the MIP Gap is defined differently for different solvers. For Gurobi the Gap is defined as follows:

$Gap[\%] = \frac{|ObjBound - ObjVal|}{|ObjVal|}$

```
In [56]: vrp.reset()
         vrp.Params.MipGap = 0.2
         vrp.optimize()
```

```
Discarded solution information
Set parameter MIPGap to value 0.2
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
MIPGap  0.2

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0xbde8918a
Variable types: 11 continuous, 1320 integer (1320 binary)
```

```
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]
Presolve removed 180 rows and 841 columns
Presolve time: 0.09s
Presolved: 260 rows, 490 columns, 1640 nonzeros
Variable types: 10 continuous, 480 integer (480 binary)
Found heuristic solution: objective 134.0000000

Root relaxation: objective 1.050909e+02, 275 iterations, 0.01 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  105.09091    0   21  134.00000  105.09091  21.6%     -    0s
H    0     0                     126.0000000  105.09091  16.6%     -    0s

Explored 1 nodes (377 simplex iterations) in 0.16 seconds (0.03 work units)
Thread count was 8 (of 8 available processors)

Solution count 2: 126 134

Optimal solution found (tolerance 2.00e-01)
Best objective 1.260000000000e+02, best bound 1.060000000000e+02, gap 15.8730%
```

Save this OFV as well. In addition, save the resulting MIP gap in percent rounded to 2 decimal places. Print the OFVs and the resulting gap. What do you notice?

```
In [57]: ofv_2 = vrp.ObjVal
         resulting_gap = round(vrp.MipGap * 100, 2)
         print(f"With gap of 0%: {ofv_1}, with gap of {resulting_gap}%: {ofv_2}")
```

```
With gap of 0%: 126.0, with gap of 15.87%: 126.0
```

The resulting gap is smaller than 20%. Gurobi terminates, when the gap reaches at least the defined gap value. Thus, the actual MIP gap in the optimization may be lower than the defined gap. The OFV with a remaining gap equals the OFV for the model that was solved to proven optimality. This phenomenon can be seen often. In some cases (especially with large instances), the optimal objective function value may even be found within a few seconds. However, proving the optimality of this solution can then take several hours. If working with large instances, it can therefore be useful to set a MIP gap or a time limit for the optimization. In many cases, we get a good (or even the optimal) solution with these limits. However, we can NOT call this solution optimal, as we have no proof of optimality.

https://support.gurobi.com/hc/en-us/community/posts/4406142530961-About-the-MIPgap

Check the solution status. What do you notice?

```
In [58]: print(f"The solution code is {vrp.Status}.")
```

```
The solution code is 2.
```

The solution status is OPTIMAL (2) even if the solver terminated with a remaining Mip gap. "The MIP solver will terminate (with an optimal result) when the gap between the lower and upper objective bound is less than MIPGap times the absolute value of the incumbent objective value." (https://www.gurobi.com/documentation/current/refman/mipgap2.html#parameter:MIPGap)

22.7 LP-Relaxation

Now, we want to compute the LP-relaxation of the model: We remove the integrality constraint on the variables X_{ijm} and Y_{im}. The LP relaxation gives us an lower bound for our problem. There are two ways for generating the LP relaxation: 1. Changing the variable definition manually. 2. Using the function relax().

More information: https://support.gurobi.com/hc/en-us/articles/360049542931-How-do-I-relax-the-integrality-conditions-in-my-model-

22.7.1 Use the manual way to compute the LP relaxation.

1. Relax the integrality constraint on the variable X_{ijm}.

```
In [59]: for i in instance.locations:
             for j in instance.locations:
                 for m in instance.tours:
                     X[i, j, m].vtype = GRB.CONTINUOUS
```

2. Relax the integrality constraint on the variable Y_{im}.

```
In [60]: for i in instance.locations:
             for m in instance.tours:
                 Y[i, m].vtype = GRB.CONTINUOUS
```

3. Optimize the model and print the LP relaxation.

```
In [61]: vrp.optimize()
         print(f"The LP relaxation is {vrp.ObjVal}.")
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
MIPGap  0.2

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0x4dbff4da
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]
Presolve removed 110 rows and 111 columns
Presolve time: 0.02s
Presolved: 330 rows, 1220 columns, 3700 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
```

```
       0    8.0000000e+00   8.090909e-01   0.000000e+00      0s
      13    4.4727273e+01   0.000000e+00   0.000000e+00      0s

Use crossover to convert LP symmetric solution to basic solution...
Crossover log...

       0 DPushes remaining with DInf 0.0000000e+00                 0s

     101 PPushes remaining with PInf 0.0000000e+00                 0s
       0 PPushes remaining with PInf 0.0000000e+00                 0s

  Push phase complete: Pinf 0.0000000e+00, Dinf 3.1086245e-15      0s

Iteration    Objective       Primal Inf.    Dual Inf.      Time
     132    4.4727273e+01   0.000000e+00   0.000000e+00      0s

Solved in 132 iterations and 0.07 seconds (0.00 work units)
Optimal objective  4.472727273e+01
The LP relaxation is 44.727272727272734.
```

4. Change the definition of the two variables back to integrality.

```
In [62]: for i in instance.locations:
             for m in instance.tours:
                 Y[i, m].vtype = GRB.BINARY
                 for j in instance.locations:
                     X[i, j, m].vtype = GRB.BINARY
```

5. Solve the model again and check if we retrieve the ofv of the non-relaxed model.

```
In [63]: vrp.optimize()
         print(f"The OFV is {vrp.ObjVal}.")
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
MIPGap  0.2

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0xbde8918a
Variable types: 11 continuous, 1320 integer (1320 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]

MIP start from previous solve produced solution with objective 126 (0.02s)
Loaded MIP start from previous solve with objective 126
```

```
Presolve removed 180 rows and 841 columns
Presolve time: 0.08s
Presolved: 260 rows, 490 columns, 1640 nonzeros
Variable types: 10 continuous, 480 integer (480 binary)

Root relaxation: interrupted, 275 iterations, 0.01 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0          -    0       126.00000  105.09091  16.6%     -    0s

Explored 1 nodes (275 simplex iterations) in 0.15 seconds (0.03 work units)
Thread count was 8 (of 8 available processors)

Solution count 1: 126

Optimal solution found (tolerance 2.00e-01)
Best objective 1.260000000000e+02, best bound 1.060000000000e+02, gap 15.8730%
The OFV is 126.0.
```

22.7.2 Use the function relax() to compute the LP relaxation.

1. create a new relaxed model.

```
In [64]: relaxed_vrp = vrp.relax()
```

2. optimize the relaxed model and print the LP relaxation.

```
In [65]: relaxed_vrp.optimize()
         print(f"The LP relaxation is {relaxed_vrp.ObjVal}.")
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
MIPGap  0.2

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0x4dbff4da
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]
Presolve removed 110 rows and 111 columns
Presolve time: 0.02s
Presolved: 330 rows, 1220 columns, 3700 nonzeros
```

```
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    8.0000000e+00   8.090909e-01   0.000000e+00      0s
      13    4.4727273e+01   0.000000e+00   0.000000e+00      0s

Use crossover to convert LP symmetric solution to basic solution...
Crossover log...

       0 DPushes remaining with DInf 0.0000000e+00                 0s

     101 PPushes remaining with PInf 0.0000000e+00                 0s
       0 PPushes remaining with PInf 0.0000000e+00                 0s

  Push phase complete: Pinf 0.0000000e+00, Dinf 3.1086245e-15      0s

Iteration    Objective       Primal Inf.    Dual Inf.      Time
     132    4.4727273e+01   0.000000e+00   0.000000e+00      0s

Solved in 132 iterations and 0.07 seconds (0.00 work units)
Optimal objective  4.472727273e+01
The LP relaxation is 44.727272727272734.
```

3. As we have created a new model, we do not need to change back the variable definition if we want to solve the non-relaxed model.

```
In [66]: vrp.optimize()
         print(f"The OFV is {vrp.ObjVal}.")
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
MIPGap  0.2

Optimize a model with 440 rows, 1331 columns and 4030 nonzeros
Model fingerprint: 0xbde8918a
Variable types: 11 continuous, 1320 integer (1320 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [4e+00, 2e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+01]
Presolved: 260 rows, 490 columns, 1640 nonzeros

Continuing optimization...


Explored 1 nodes (275 simplex iterations) in 0.03 seconds (0.00 work units)
Thread count was 8 (of 8 available processors)

Solution count 1: 126
```

```
Optimal solution found (tolerance 2.00e-01)
Best objective 1.260000000000e+02, best bound 1.060000000000e+02, gap 15.8730%
The OFV is 126.0.
```

22.8 Model with the data from instance generator

22.8.1 Model with one instance

We start with loading a dataset created with the instance generator. Let's take a dataset with 40 locations.

```
In [67]: instance_module_vrp.generate_instances(nb_instances=5, nb_locations=40,␣
 ↪capacity=5)

         instance_large = pickle.load(open("../data/vrp_data/i40_b5_1.vrp", "rb"))
```

Create a model with this instance data. Thanks to our functions, this can be done easily.

```
In [68]: large_vrp = gp.Model()

         X, Y = build_vrp(large_vrp, instance_large)
```

Let's check if this worked correctly: Print the number of binary variables. With 40 locations, we should have $40 \cdot 40 \cdot 39 = 62400$ X variables and $40 \cdot 39 = 1560$ Y variables, resulting to 63960 binary variables. Remember that Gurobi uses a lazy update approach. This means the model is not updated after each variable/constraint that is added. An update of the model is only performed if we call update(), write() or optimize(). So if we want to query the number of binary variables before we have solved the model or wrote an LP file, we need to call update.

```
In [69]: large_vrp.update()
         print(large_vrp.NumBinVars)
```

```
63960
```

We are now able to solve the model as usual. However, if you do so, you will see that it takes quiet some time for this instance size. Thus, for large instances, it is usually good to set a time limit. As an example, set the time limit to 20 seconds.

(Of course, the time limit always depends on the model and the purpose of solving. For some problems a time limit of several hours or days is fine, while other problems should be solved within a few seconds!)

```
In [70]: large_vrp.Params.TimeLimit = 20
         # large_vrp.optimize()
```

```
Set parameter TimeLimit to value 20
```

22.8.2 Solving a model multiple times with different instances

Now, we want to solve the model multiple times for different capacities (as in Helber (2020), p. 243). Let's use a smaller instance, e.g. 10 locations. Generate one instance dataset with 10 locations with the instance generator. We just use an arbitrary capacity. We will change the capacity in the next step.

After we have generated the instance, we define a list with the considered capacities: 5, 6, 7, 10 and 20.

```
In [71]: capacities = [5, 6, 7, 10, 20]
```

```
In [72]: instance_module_vrp.generate_instances(nb_instances=1, nb_locations=10,␣
 ↪capacity=5)

         instance = pickle.load(open("../data/vrp_data/i10_b5_1.vrp", "rb"))
```

Loop over all capacities. For each capacity, overwrite the capacity of the instance loaded. Than create the model by using the created function and solve the model. Print the objective function value and the capacity for each solved model.

```
In [73]: for b in capacities:

             # define an empty model
             vrp = gp.Model()

             # suppress logging to console
             vrp.Params.LogToConsole = 0

             # overwrite the capacity
             instance.capacity = b

             # build the model
             X, Y = build_vrp(vrp=vrp, instance=instance)

             # optimize the model and print the OFV
             vrp.optimize()
             print(f"For a capacity of {b}, the OFV is {vrp.ObjVal}.")
```

```
Set parameter LogToConsole to value 0
```

```
For a capacity of 5, the OFV is 125.28999999999999.
Set parameter LogToConsole to value 0
For a capacity of 6, the OFV is 107.43.
Set parameter LogToConsole to value 0
For a capacity of 7, the OFV is 85.84.
Set parameter LogToConsole to value 0
For a capacity of 10, the OFV is 63.6.
Set parameter LogToConsole to value 0
For a capacity of 20, the OFV is 39.15.
```

22.8.3 Another way to solve a model multiple times

In the example above, we have used a function for creating the model. However, we still build a complete new model each time we change the capacity, even though a large portion of the model remains unchanged. It is also possible to just update constraints of the model (or delete a constraint and add a new one.)

```
In [74]: # define an empty model and suppress logging
         vrp = gp.Model()
         vrp.Params.LogToConsole = 0

         # build model
         X, Y = build_vrp(vrp=vrp, instance=instance)
         vrp.update()

         for b in capacities:
             # set capacity
             instance.capacity = b

             # we update the first constraint with the defined capacity
             for m in instance.tours:
                 # for every tour m (the constraint is defined for each tour), we get␣
 ↪the constraint by its name
                 constraint = vrp.getConstrByName(f"capacity[{m}]")

                 # we change the right hand side with .rhs to the defined capacity
                 constraint.rhs = instance.capacity

             # solve model
             vrp.optimize()

             print(f"For a capacity of {b}, the OFV is {vrp.ObjVal}.")
```

```
Set parameter LogToConsole to value 0
For a capacity of 5, the OFV is 125.28999999999999.
For a capacity of 6, the OFV is 107.43.
For a capacity of 7, the OFV is 85.83999999999999.
For a capacity of 10, the OFV is 63.6.
For a capacity of 20, the OFV is 39.15.
```

22.9 Callbacks

Callbacks are mainly used for two purposes: - report on the progress of the optimization - modify the behavior of the Gurobi solver

In this example, we will see how to report the solution progress. The TSP example will give you insights on how to modify the behavior of the solver.

More information on callbacks: https://www.gurobi.com/documentation/9.5/examples/cb_s.html

Callback Codes: https://www.gurobi.com/documentation/current/refman/cb_codes.html

We will first define a callback for tracking the solution process. We want to get the current objective function value and the time if we found a new best solution.

```
In [75]: def time_tracking_callback(model, where):
             if where == GRB.Callback.MIPSOL:

                 # get the current objective of the node
                 obj = model.cbGet(GRB.Callback.MIPSOL_OBJ)

                 # get the current runtime
```

```
                time = model.cbGet(GRB.Callback.RUNTIME)

                # save the objective and the time as a tuple in the previously defined␣
↪variable
                model._sol_process.append((obj, time))

                # print the information to the console
                print(f"New solution with objective {obj} found after {time} seconds")
```

```
In [76]: instance = pickle.load(open("../data/vrp_data/i10_b5_1.vrp", "rb"))

         vrp_callback = gp.Model()

         X, Y = build_vrp(vrp_callback, instance)

         # define a list variable to track the solution process
         vrp_callback._sol_process = []

         # set the time limit to 100 seconds
         vrp_callback.Params.TimeLimit = 100

         # optimize the model with the callback
         vrp_callback.optimize(time_tracking_callback)

         for obj, time in vrp_callback._sol_process:
             print(f"Time in sec: {time:.2f}, obj: {obj:.2f}")
```

```
Set parameter TimeLimit to value 100
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.5 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
TimeLimit  100

Optimize a model with 360 rows, 1000 columns and 3024 nonzeros
Model fingerprint: 0xd7fc5c99
Variable types: 10 continuous, 990 integer (990 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+01]
  Objective range  [1e+00, 1e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 9e+00]
```

```
Presolve removed 135 rows and 604 columns
Presolve time: 0.06s
Presolved: 225 rows, 396 columns, 1350 nonzeros
Variable types: 9 continuous, 387 integer (387 binary)
New solution with objective 132.81 found after 0.08422303199768066 seconds
Found heuristic solution: objective 132.8100000
```

```
Root relaxation: objective 1.058860e+02, 268 iterations, 0.01 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  105.88600    0   16  132.81000  105.88600  20.3%     -    0s
New solution with objective 125.29 found after 0.11640715599060059 seconds
H    0     0                     125.2900000  105.88600  15.5%     -    0s
     0     0  110.87000    0   38  125.29000  110.87000  11.5%     -    0s
     0     0  110.87000    0   23  125.29000  110.87000  11.5%     -    0s
     0     0  110.87000    0   18  125.29000  110.87000  11.5%     -    0s
     0     0  112.03200    0   14  125.29000  112.03200  10.6%     -    0s
     0     0  116.08000    0   12  125.29000  116.08000  7.35%     -    0s
     0     0  116.08000    0   12  125.29000  116.08000  7.35%     -    0s
     0     0  116.08000    0   12  125.29000  116.08000  7.35%     -    0s
     0     0  116.08000    0   12  125.29000  116.08000  7.35%     -    0s
     0     0  116.97500    0   18  125.29000  116.97500  6.64%     -    0s
     0     0  122.93000    0   16  125.29000  122.93000  1.88%     -    0s
     0     0  122.93000    0   22  125.29000  122.93000  1.88%     -    0s
     0     0  122.93000    0   12  125.29000  122.93000  1.88%     -    0s
     0     0  124.16000    0   18  125.29000  124.16000  0.90%     -    0s
     0     0  124.72000    0   12  125.29000  124.72000  0.45%     -    0s
     0     0  124.72000    0   19  125.29000  124.72000  0.45%     -    0s
     0     0 infeasible    0       125.29000  125.29000  0.00%     -    0s
     0     0 infeasible    0       125.29000  125.29000  0.00%     -    0s

Cutting planes:
  Implied bound: 4
  Clique: 4
  MIR: 3
  RLT: 1

Explored 1 nodes (677 simplex iterations) in 0.29 seconds (0.04 work units)
Thread count was 8 (of 8 available processors)

Solution count 2: 125.29 132.81

Optimal solution found (tolerance 1.00e-04)
Best objective 1.252900000000e+02, best bound 1.252900000000e+02, gap 0.0000%

User-callback calls 459, time in user-callback 0.02 sec
Time in sec: 0.08, obj: 132.81
Time in sec: 0.12, obj: 125.29
```

Chapter 23

Gurobi Logfiles

23.1 Creating a Logfile

```
In [1]: import gurobipy as gp
        from gurobipy import GRB
        import os
```

Let's create our folder for the logfile.

```
In [2]: path = "../results/lecture12"
        if not os.path.exists(path):
            os.makedirs(path)
```

Define the simple model 1 as in lecture 10.

```
In [3]: simple_model_1 = gp.Model()

        # Decision variables
        X1 = simple_model_1.addVar(vtype=GRB.CONTINUOUS, name="X1", lb=0)
        X2 = simple_model_1.addVar(vtype=GRB.CONTINUOUS, name="X2", lb=0)
        Y = simple_model_1.addVar(vtype=GRB.INTEGER, name="Y", lb=0, ub=7)

        # Objective function
        simple_model_1.setObjective((6 * X1 + 5 * X2 + 3 * Y), GRB.MINIMIZE)

        # Constraints
        # First constraint
        simple_model_1.addConstr((X1 + X2 >= 6), "first")

        # Second constraint
        simple_model_1.addConstr((2 * X1 + 3 * X2 - 4 * Y <= 0), "second")
```

```
Restricted license - for non-production use only - expires 2026-11-23
```

```
Out[3]: <gurobi.Constr *Awaiting Model Update*>
```

Set the path for the logfile and optimize the model.

```
In [4]: # set LogFile
        simple_model_1.Params.LogFile = f"{path}/simple_model_1.log"

        # optimize
        simple_model_1.optimize()
```

```
Set parameter LogFile to value "../results/lecture12/simple_model_1.log"
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (linux64 - "Ubuntu 22.04.4 LTS")

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 3 columns and 5 nonzeros
Model fingerprint: 0x1f88faca
Variable types: 2 continuous, 1 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 4e+00]
  Objective range  [3e+00, 6e+00]
  Bounds range     [7e+00, 7e+00]
  RHS range        [6e+00, 6e+00]
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 5 nonzeros
Variable types: 2 continuous, 1 integer (0 binary)
Found heuristic solution: objective 51.0000000

Root relaxation: objective 4.350000e+01, 2 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0   43.50000    0    1   51.00000   43.50000  14.7%     -    0s
H    0     0                       45.0000000   43.50000  3.33%     -    0s
*    0     0               0       44.0000000   44.00000  0.00%     -    0s

Explored 1 nodes (4 simplex iterations) in 0.04 seconds (0.00 work units)
Thread count was 8 (of 8 available processors)

Solution count 3: 44 45 51

Optimal solution found (tolerance 1.00e-04)
Best objective 4.400000000000e+01, best bound 4.400000000000e+01, gap 0.0000%
```

23.2 Structure of a Logfile

Information about logfiles can be found at https://docs.gurobi.com/projects/optimizer/en/current/concepts/logging/mip.htm

One (other) example for a logfile:

```
Gurobi Optimizer version 10.0.2 build v10.0.2rc0 (win64)

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 66 rows, 90 columns and 205 nonzeros
Model fingerprint: 0x3d8a0e2e
Variable types: 60 continuous, 30 integer (30 binary)
Coefficient statistics:
  Matrix range     [1e+00, 4e+02]
  Objective range  [3e+00, 5e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [5e+00, 4e+02]
Presolve removed 18 rows and 24 columns
Presolve time: 0.00s
Presolved: 48 rows, 66 columns, 160 nonzeros
Variable types: 40 continuous, 26 integer (26 binary)
Found heuristic solution: objective 4495.0000000

Root relaxation: objective 1.332276e+03, 30 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 1332.27562    0   16 4495.00000 1332.27562  70.4%     -    0s
H    0     0                     3940.0000000 1332.27562  66.2%     -    0s
                                      ⋮
H   42    25                     2500.0000000 2351.07765  5.96%   8.0    0s
H  112     2                     2495.0000000 2411.65913  3.34%   7.0    0s

Cutting planes:
  Gomory: 4
  Cover: 4
  Implied bound: 9
  MIR: 19
  Flow cover: 15
  Flow path: 2

Explored 125 nodes (1002 simplex iterations) in 0.09 seconds (0.02 work units)
Thread count was 8 (of 8 available processors)

Solution count 7: 2495 2500 2570 ... 4495

Optimal solution found (tolerance 1.00e-04)
Best objective 2.495000000000e+03, best bound 2.495000000000e+03, gap 0.0000%
```

Logfile in Detail:

1. General Information:


```
Gurobi Optimizer version 10.0.2 build v10.0.2rc0 (win64)

CPU model: Intel(R) Core(TM) i7-10610U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
```


- Gurobi version used (10.0.2)

- Processor of the computer used
- Threads and cores
  - Core: physical processor unit within a central processing unit (CPU)
  - A physical core can appear as several logical processors (here as two)
  - Threads: Number of tasks to be performed simultaneously (e.g. for Gurobi: number of nodes examined simultaneously).
  - The number of threads can be set using parameters (important for comparisons).

2. Summarized information on the model

```
Optimize a model with 66 rows, 90 columns and 205 nonzeros
Model fingerprint: 0x3d8a0e2e
Variable types: 60 continuous, 30 integer (30 binary)
```

- Size of the model
- Fingerprint: hash value based on the attributes of all components of the model (ID)
- Information about variables

3. Statistics on the coefficients

```
Coefficient statistics:
  Matrix range     [1e+00, 4e+02]
  Objective range  [3e+00, 5e+01]
  Bounds range     [1e+00, 1e+00]
  RHS range        [5e+00, 4e+02]
```

- E.g. `Objective range`: All coefficients of the objective function are between 3 and 50

4. Presolve

```
Presolve removed 18 rows and 24 columns
Presolve time: 0.00s
Presolved: 48 rows, 66 columns, 160 nonzeros
Variable types: 40 continuous, 26 integer (26 binary)
```

- In the presolve stage, 18 rows and 24 columns could be removed in 0.00 seconds.
- The model now contains 48 rows and 66 columns.
- This leaves 40 real-valued and 26 integer variables.

5. Heuristic solution and relaxed root node

```
Found heuristic solution: objective 4495.0000000

Root relaxation: objective 1.332276e+03, 30 iterations, 0.00 seconds (0.00 work units)
```

- Using a heuristic, an integer solution with an objective function value of 4495 was found.
- The relaxed root node is then solved (objective function value: 1332.3)
  - In some cases, more time is required for this and therefore detailed logging is performed.

6. Branch-and-Bound

```
    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 1332.27562    0   16 4495.00000 1332.27562  70.4%     -    0s
H    0     0                     3940.0000000 1332.27562  66.2%     -    0s
H    0     0                     2790.0000000 1332.27562  52.2%     -    0s
     0     0 2091.84645    0   14 2790.00000 2091.84645  25.0%     -    0s
```

- `Nodes`
  - `Expl`: Number of nodes that were examined
  - `Unexpl`: Number of nodes that have not yet been examined
  - `H` or `*`: A new permissible solution was found
    - `H` = via a heuristic
    - `*` = via branching
  - *Notes:*
    - Number of examined nodes often remains at 0 for a longer period of time at the beginning
    - The root node is processed during this time.
    - Cutting planes are created and root nodes are used to reduce the size of the branch-and-bound.
- `Current Node`
  - `Obj`: Target function value of the relaxation
  - `Depth`: Depth of the node in the branch-and-bound
  - `IntInf`: Number of integer variables with non-integer values
- `Objective Bounds`:
  - `Incumbent`: best known objective function value for a feasible solution
  - `BestBd`: lower bound for the objective function value
  - `Gap`: relative difference between the two bounds
- `Work`:
  - `It/Node`: average number of simplex iterations per node
  - `Time`: elapsed time - by default, a log line is output at least every 5 seconds (can be set via the DisplayInterval parameter)

7. Cutting Planes

```
Cutting planes:
  Gomory: 4
  Cover: 4
  Implied bound: 9
  MIR: 19
  Flow cover: 15
  Flow path: 2
```

- Number of applied cutting planes in the LP-relaxation at the end of the MIP search

8. Final Results

```
Explored 125 nodes (1002 simplex iterations) in 0.09 seconds (0.02 work units)
Thread count was 8 (of 8 available processors)

Solution count 7: 2495 2500 2570 ... 4495

Optimal solution found (tolerance 1.00e-04)
Best objective 2.495000000000e+03, best bound 2.495000000000e+03, gap 0.0000%
```

- Number of nodes examined with time specification (125 nodes in 0.09 seconds)
- Number of threads used (8 in the example)
- Number of solutions found in the solution process (7 solutions)
  - If there are more than 10 solutions, only the best 10 are shown
- The gap is 0% so that an optimal solution has been found.

23.3 Gurobi LogTools

First, we need to install `grblogtools` via `pip` and import it afterwards as `glt`.

```
In [5]: import grblogtools as glt
```

```
In [6]: results = glt.parse(f"{path}/simple_model_1.log")
        summary = results.summary()
        nodelog_progress = results.progress("nodelog")

        display(nodelog_progress)
```

```
    CurrentNode  RemainingNodes   Obj  Depth  IntInf  Incumbent  BestBd  \
0             0             0.0  43.5    0.0     1.0       51.0    43.5
1             0             0.0   NaN    NaN     NaN       45.0    43.5
2             0             0.0   NaN    0.0     NaN       44.0    44.0
3             1             NaN   NaN    NaN     NaN       44.0    44.0
4             0             0.0  43.5    0.0     1.0       51.0    43.5
5             0             0.0   NaN    NaN     NaN       45.0    43.5
6             0             0.0   NaN    0.0     NaN       44.0    44.0
7             1             NaN   NaN    NaN     NaN       44.0    44.0
8             0             0.0  43.5    0.0     1.0       51.0    43.5
9             0             0.0   NaN    NaN     NaN       45.0    43.5
10            0             0.0   NaN    0.0     NaN       44.0    44.0
11            1             NaN   NaN    NaN     NaN       44.0    44.0
12            0             0.0  43.5    0.0     1.0       51.0    43.5
13            0             0.0   NaN    NaN     NaN       45.0    43.5
14            0             0.0   NaN    0.0     NaN       44.0    44.0
15            1             NaN   NaN    NaN     NaN       44.0    44.0

       Gap  ItPerNode  Time NewSolution  \
0   0.1470        NaN  0.00         NaN
1   0.0333        NaN  0.00           H
2   0.0000        NaN  0.00           *
3   0.0000        NaN  0.08         NaN
4   0.1470        NaN  0.00         NaN
5   0.0333        NaN  0.00           H
6   0.0000        NaN  0.00           *
7   0.0000        NaN  0.13         NaN
8   0.1470        NaN  0.00         NaN
9   0.0333        NaN  0.00           H
10  0.0000        NaN  0.00           *
11  0.0000        NaN  0.08         NaN
12  0.1470        NaN  0.00         NaN
13  0.0333        NaN  0.00           H
14  0.0000        NaN  0.00           *
15  0.0000        NaN  0.04         NaN

                             LogFilePath  LogNumber  Seed Version
0   ../results/lecture12/simple_model_1.log          1     0  12.0.0
1   ../results/lecture12/simple_model_1.log          1     0  12.0.0
2   ../results/lecture12/simple_model_1.log          1     0  12.0.0
3   ../results/lecture12/simple_model_1.log          1     0  12.0.0
4   ../results/lecture12/simple_model_1.log          2     0  12.0.0
```

```
5   ../results/lecture12/simple_model_1.log          2     0  12.0.0
6   ../results/lecture12/simple_model_1.log          2     0  12.0.0
7   ../results/lecture12/simple_model_1.log          2     0  12.0.0
8   ../results/lecture12/simple_model_1.log          3     0  12.0.0
9   ../results/lecture12/simple_model_1.log          3     0  12.0.0
10  ../results/lecture12/simple_model_1.log          3     0  12.0.0
11  ../results/lecture12/simple_model_1.log          3     0  12.0.0
12  ../results/lecture12/simple_model_1.log          4     0  12.0.0
13  ../results/lecture12/simple_model_1.log          4     0  12.0.0
14  ../results/lecture12/simple_model_1.log          4     0  12.0.0
15  ../results/lecture12/simple_model_1.log          4     0  12.0.0
```

Chapter 24

Traveling Salesperson Problem

24.1 Model Formulation

Abstract Mathematical Model

Indices and quantities - $i, j \in \mathcal{I} = \{1, \ldots, I\}$: Locations with location 1 as depot
Parameters - d_{ij}: distance between locations
Decision variables - $X_{ij} \in \{0, 1\}$: 1 if driving from location i to location j and 0 otherwise

$$\min \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} d_{ij} \cdot X_{ij}$$

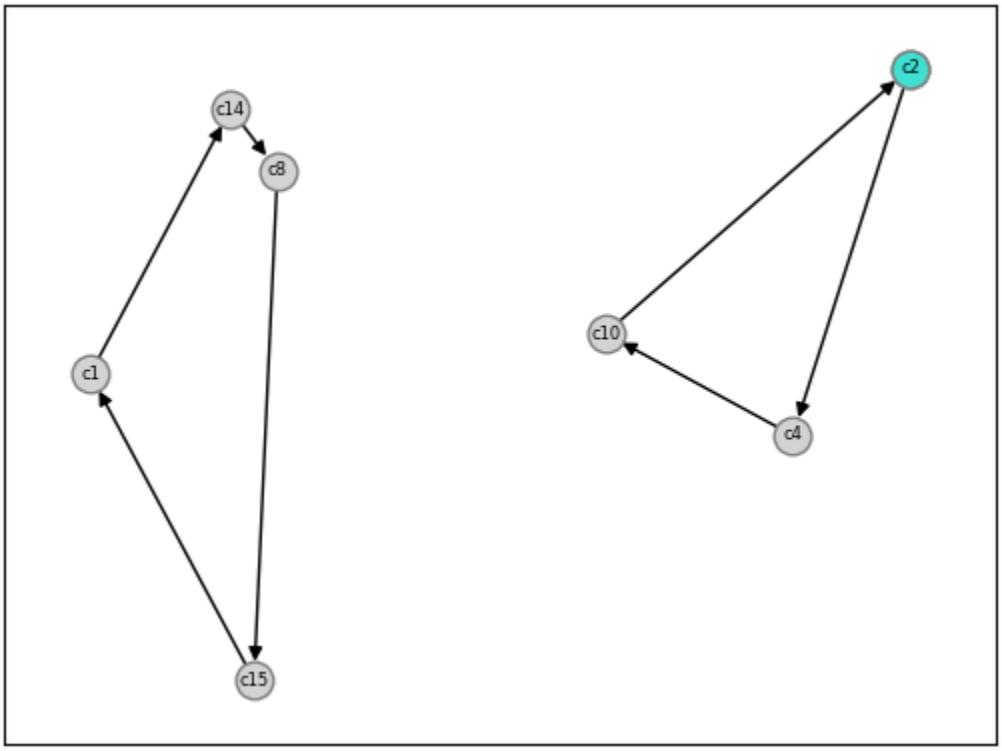
s. t.

$$\sum_{j \in \mathcal{J}} X_{ij} = 1 \qquad \forall\ i \in \mathcal{I}$$

$$\sum_{i \in \mathcal{I}} X_{ij} = 1 \qquad \forall\ j \in \mathcal{J}$$

Avoidance of subtours without depot

Subtours without depot look like this:



24.2 Subtour Elimination Constraints

24.2.1 Model Formulation I (Miller–Tucker–Zemlin formulation)

Creation of an auxiliary variable π_i for each location i with $i = 2, ..., I$ (as in the VRP example).

$$\min \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} d_{ij} \cdot X_{ij}$$

s. t.

$$\begin{aligned}
\sum_{j \in \mathcal{J}} X_{ij} &= 1 && \forall\ i \in \mathcal{I} \\
\sum_{i \in \mathcal{I}} X_{ij} &= 1 && \forall\ j \in \mathcal{J} \\
\pi_i - \pi_j + I \cdot X_{ij} &\leq I - 1 && \forall\ i, j = 2, ..., I, i \neq j
\end{aligned}$$

Example with 3 locations and the depot at location 1:

$$\begin{aligned}
\pi_2 - \pi_3 + 3 \cdot X_{2,3} &\leq 3 - 1 \\
\pi_3 - \pi_2 + 3 \cdot X_{3,2} &\leq 3 - 1
\end{aligned}$$

With $X_{3,2} = 1$ and $X_{2,3} = 1$:

$$\begin{aligned}
\pi_2 - \pi_3 &\leq -1 \\
\pi_3 - \pi_2 &\leq -1
\end{aligned}$$

24.2.2 Model Formulation II (Dantzig–Fulkerson–Johnson formulation)

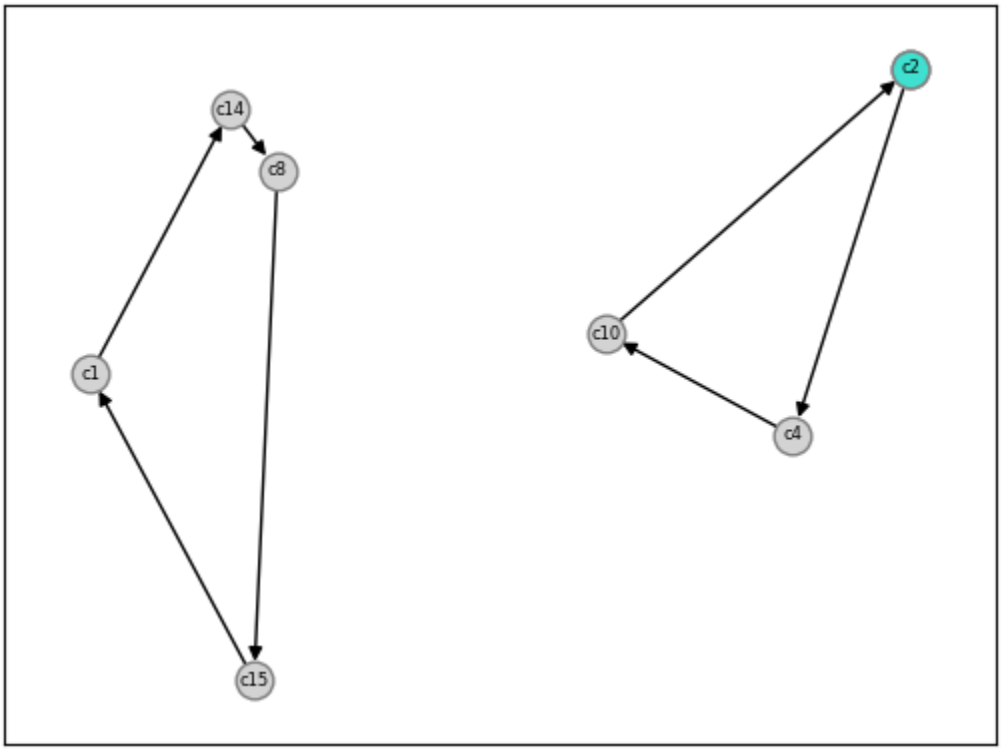
At least one arrow from each subset of locations must lead into the set of other locations. Thus, we need to generate all possible subsets $\mathcal{U}$ of the locations for which $2 \leq |\mathcal{U}| \leq \frac{|\mathcal{I}|}{2}$.

$$\min \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} d_{ij} \cdot X_{ij}$$

s. t.

$$\begin{aligned}
\sum_{j \in \mathcal{J}} X_{ij} &= 1 && \forall\, i \in \mathcal{I} \\
\sum_{i \in \mathcal{I}} X_{ij} &= 1 && \forall\, j \in \mathcal{J} \\
\sum_{i \in \mathcal{U}, j \in \mathcal{I} \setminus \mathcal{U}} X_{ij} &\geq 1 && \forall\, \mathcal{U} \subset \mathcal{I}, 2 \leq |\mathcal{U}| \leq \frac{|\mathcal{I}|}{2}
\end{aligned}$$

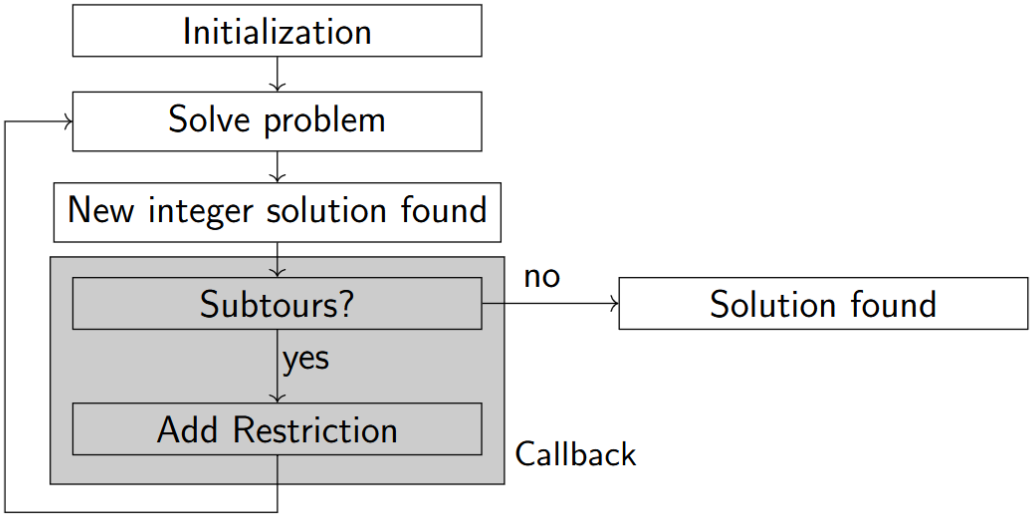
Example with locations $c1, c2, c3, c4$
$\mathcal{U} = \{\{c1, c2\}, \{c1, c3\}, \{c1, c4\}, \{c2, c3\}, \{c2, c4\}, \{c3, c4\}\}$
Example from above:



Restriktion fulfilled for the subset $\{c2, c4\}$ Restriktion not fulfilled for the subset $\{c2, c4, c10\}$

24.3 Using Callbacks for the Model Formulation II

- Problem: A large number of subsets and therefore restrictions have to be created if the number of locations is large.
- Alternatively, these restrictions can initially be omitted. This means, we start with solving the model with just the restrictions (1) and (2).
- If a new solution is found in Gurobi, the solution is checked for subtours.
- If a subtour is found, a constraint can be added to avoid this specific subtour. These restrictions can be added in Gurobi with a callback using a lazy constraints.



Lazy Constraints with Callbacks

- Lazy constraints in Gurobi: Constraints that are mandatory for the model. Without these restrictions, the model would be incorrect.
- They are only added in the callback if they are violated → Goal: Save computing time
- Parameter for lazy constraints has to be set: `LazyConstraint=1`
- If a subtour $\mathcal{C}$ consisting of the cities i is found in the callback, the following restriction (lazy constraints) is added:

$$\sum_{i \in \mathcal{C}, j \in \mathcal{I} \setminus \mathcal{C}} X_{ij} \geq 1$$