Python for Decision Analysis and Optimization

Pieter Kleer

Table of contents

Chapter 1

Welcome

In this online book we will learn some elementary Python programming tools, as well as the implementation of (linear) optimization problems in Python.

To install Python we will use the Anaconda distribution, which contains all relevant Python software that we will need. In particular, we will use the Spyder editor, or integrated development environment (IDE), to construct our Python programs.

ANACONDA.

Figure 1.1: Anaconda

To solve (linear) optimization problems we will use Gurobi, a state-of-the-art software package for solving optimization problems that is used extensively in businesses and academia. We will use this software within Python.

GUROBI
OPTIMIZATION

Figure 1.2: Gurobi

In the next chapter we describe how you can install the relevant software, after which we explain some Python basics in Chapter 3. In Chapter 4, we will explain how to implement optimization problems in Python and solve them with the Gurobi solver.

Chapter 2

Software

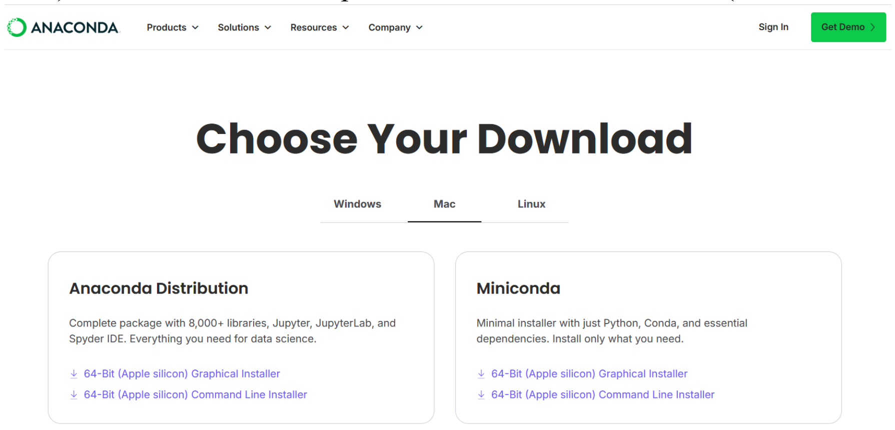
In this chapter we will learn how to install Python and run our very first command. You are of course also free to use your own Python installation. In that case it is best to check the Gurobi documentation to see how to properly install this program.

2.1 Anaconda

The easiest way to install Python and the Spyder editor at once is by installing Anaconda, which is a software package that includes Python, Spyder, and various other software applications. In the block below, you can find an outline for installing Anaconda.

Anaconda installation steps

1. Go to https://www.anaconda.com/download
2. Click on "Get started" and sign up with an e-mail address of your choice; verify your e-mail address as instructed.
3. Go back to the above Anaconda webpage (either by signing in or via the verification e-mail) and download the *Graphical Installer* installation file (both for MacOS and Windows).



Choose Your Download

Anaconda Distribution

Complete package with 8,000+ libraries, Jupyter, JupyterLab, and Spyder IDE. Everything you need for data science.

64-Bit (Apple silicon) Graphical Installer

64-Bit (Apple silicon) Command Line Installer

Miniconda

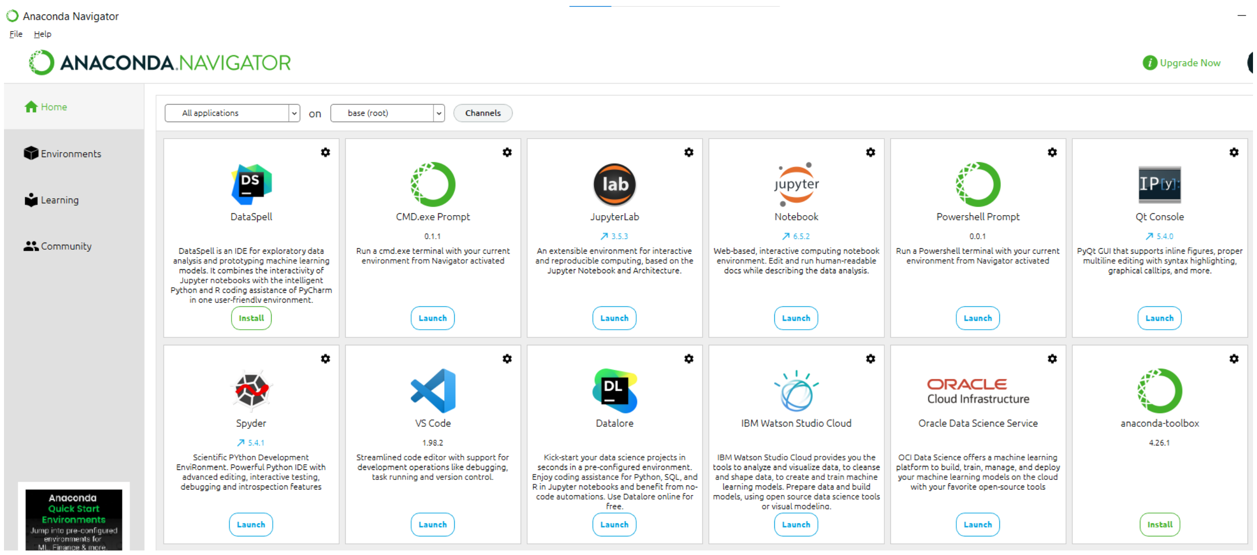
Minimal installer with just Python, Conda, and essential dependencies. Install only what you need.

64-Bit (Apple silicon) Graphical Installer

64-Bit (Apple silicon) Command Line Installer

4. After downloading the installation file, install it. Follow the installation wizard and keep all the default options during installation.

After installation, you can see all the newly installed applications using the *Anaconda Navigator*. You can open the navigator by searching for the Anaconda Navigator in the Start menu (on Windows) or the equivalent on MacOS.



DataSpell is an IDE for exploratory data analysis and prototyping machine learning models. It combines the interactivity of Jupyter notebooks with the intelligent Python and R coding assistance of PyCharm in one user-friendly environment.

Run a cmd.exe terminal with your current environment from Navigator activated

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Run a Powershell terminal with your current environment from Navigator activated

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Streamlined code editor with support for development operations like debugging, task running and version control.

Kick-start your data science projects in seconds in a pre-configured environment. Enjoy coding assistance for Python, SQL, and R in Jupyter notebooks and benefit from no-code automations. Use Datalore online for free.

IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.

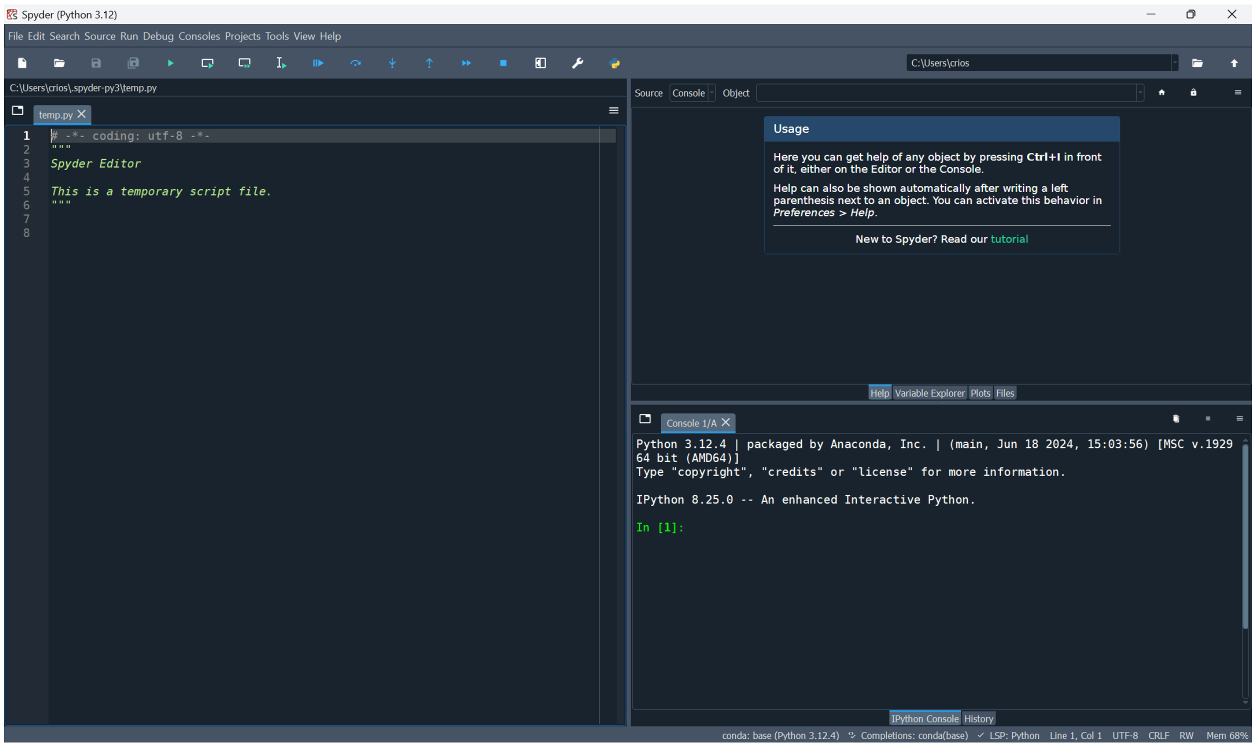
OCI Data Science offers a machine learning platform to build, train, manage, and deploy your machine learning models on the cloud with your favorite open-source tools

Figure 2.1: Anaconda Navigator

2.2 Spyder

To create Python code and scripts, we will be using the Spyder application, which you can open by clicking on *Launch* in the Anaconda Navigator or by searching for the Spyder application on your computer directly. We note that there are many other applications that you can use to create Python code with (such as Visual Studio Code and Jupyter Notebook; two other applications also installed in the Anaconda distribution).

Once you have opened Spyder, you should see an application that looks like this:


Spyder (Python 3.12)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\crios
C:\Users\crios\.spyder-py3\temp.py
temp.py
Source Console Object
Help Variable Explorer Plots Files
Console 1/A
IPython Console History
conda: base (Python 3.12.4) Completions: conda(base) LSP: Python Line 1, Col 1 UTF-8 CRLF RW Mem 68%

```python
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

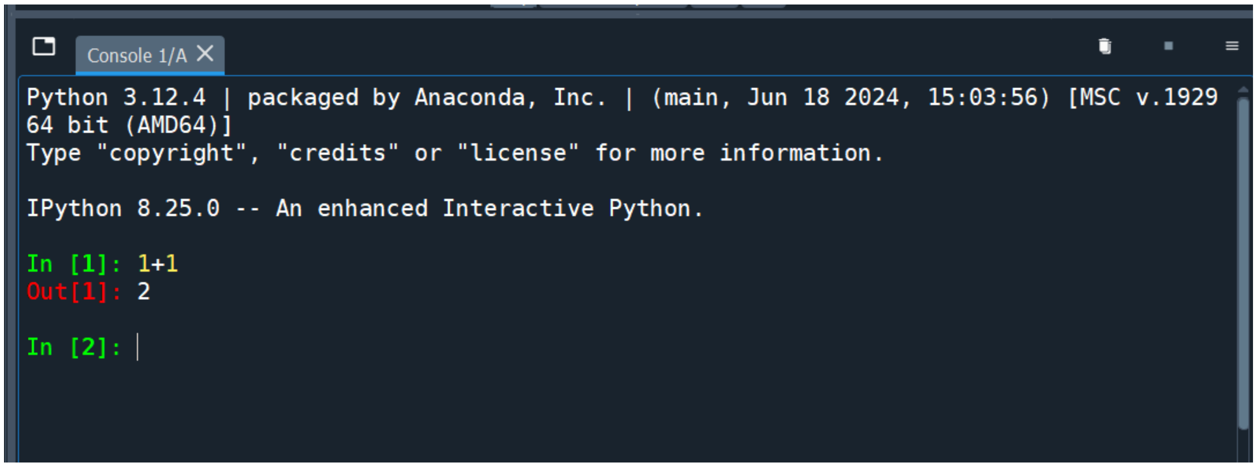
New to Spyder? Read our tutorial

```
Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MSC v.1929
64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.25.0 -- An enhanced Interactive Python.

In [1]:
```

Figure 2.2: Spyder

2.2.1 Python Console

In the bottom right pane you see a console with IPython. IPython is short for *Interactive Python*. We can type Python commands into this console and see the output directly. To find $1 + 1$ in Python, we can use the command `1+1`, similar to how we would do it in Excel or in the Google search engine. Let's try this out in the console. First, click on the console to move the cursor there. Then type `1+1` and press `Enter`. We will see the output `2` on the next line next to a red `Out [1]`:



```
Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MSC v.1929
64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.25.0 -- An enhanced Interactive Python.

In [1]: 1+1
Out[1]: 2

In [2]:
```

Figure 2.3: IPython Console

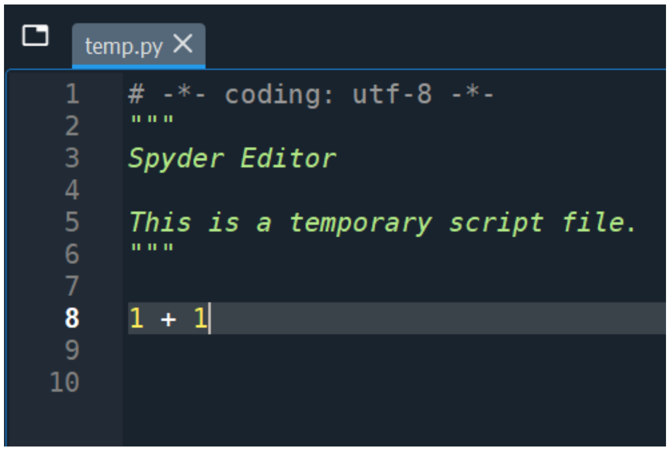
The red `Out [1]` means this is the output from the first line of input (after the green `In [1]`). The second command will have input `In [2]` and output `Out [2]`.

2.2.2 Python Scripts

Typing commands directly into the IPython console is fine if all you want to do is try out a few different simple commands (like we do at the start of the next chapter). However, when working on a project you will often be executing many commands. If you were to do all of this in the interactive console it would be very easy to lose track of what you are doing. It would also be very easy to make mistakes.

Writing your commands in Python scripts is a solution to this problem. A Python script is a text file with a `.py` extension where you can write all of your commands in the order you want them run. You can then get Spyder to run the entire file of commands (or only a part of them).

In Spyder, in the left pane you see a file open called `temp.py` when you start up Spyder. This is an example Python script. We can ignore what is written in the first 6 lines of the script. In fact, you can remove these lines if you want. We can add our `1 + 1` command to the bottom of the script like this and save it:

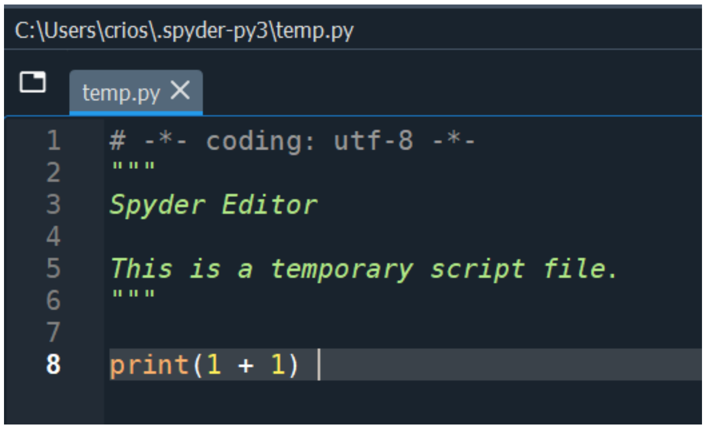


```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

1 + 1
```

Figure 2.4: Python Script

In the Toolbar there are several ways to run this command from the script. For example, you can run the entire file, or run only the current line or selected area. If the cursor is on the line with `1 + 1` and we press the "Run selection or current line" button, then we will see the command and output appear in the IPython console, just like how we typed it there before. Using the script, however, we have saved and documented our work.

If you try run the entire file, you will see `runfile('...')` in the IPython console with the `...` being the path to the Python script you are running. However, you don't see a `2` in the output. This is because when running an entire file, Python does not show the output of each line being run. To see the output of any command we need to put it inside the `print()` function. We can change our line to `print(1 + 1)` to see the output when running the entire file:



```
C:\Users\crios\.spyder-py3\temp.py
temp.py
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

print(1 + 1)
```

Figure 2.5: Using the print() function

When you run the entire file you should now see a `2` below the `runfile('...')` command. We now know how to write and run Python scripts! In the next chapter we will learn more Python commands.

2.2.3 Code snippets in this online book

In this book, we won't always show screenshots like we did above. Instead we will show code snippets in boxes like this:

```
1 + 1
```

```
2
```

The top part is code and the bottom part in gray is the output of the executed code. There is a small clipboard icon on the right which you can use to copy the code to paste into your own Python file to be able to experiment with it yourself.

2.3 Gurobi installation

Based on the above Anaconda installation, we will next explain how to install Gurobi, which is a software package that can solve (linear) optimization problems via Python. We assume in the below that you already have Anaconda/Python installed on your system as in the above steps.

If you get stuck anywhere in the installation process, then you can find more information on Gurobi's Quick Start Guide.

2.3.1 Register at Gurobi

Visit the Gurobi website and click on the register button. Open the registration form and make sure that you select the *Academic* account type, and select *Student* for the academic position. Register with your TIAS e-mail account.

2.3.2 Download Gurobi

Download Gurobi from the website. Note that you need to login with your Gurobi account before you can download Gurobi. Select the distribution that corresponds to your system (e.g., Windows or macOS), and select the regular "Gurobi Optimizer", not any of the AMPL variations. Unless mentioned otherwise, download the most recent version.

2.3.3 Install Gurobi

Run the installer and follow the installation steps. At some point, the installer may ask whether to add Gurobi to your execution path. This is probably useful to accept.

2.3.4 Gurobi license

You cannot use Gurobi without a license, so you need to apply for a license. As a student you can request a free academic license; take the **Named-User Academic** one. After you have obtained the license, you need to activate it for your Gurobi installation. If you open the license details on the Gurobi website, you can see what you need to do: open a Command Prompt (Windows) or Terminal (MacOS) and run the `grbgetkey` command with the code that corresponds to your license. This command will create your license file. Make sure that you remember where the license file is saved. The default location suggested by your device is probably the best choice.

Note that the `grbgetkey` command will check that you are on an academic domain, so you need to perform this step on the **university network** (possibly via a VPN connection). Once installed correctly, you can also run Gurobi without an active VPN connection.

2.3.5 Using Gurobi in Python

Gurobi should now be correctly installed, but we also want to be able to use it from Python. Therefore, we need to install Gurobi's Python package. Open the Anaconda prompt application (can also be found in the Anaconda Navigator) and run the following commands one by one (first line + Enter, second line + Enter).

- `conda config --add channels http://conda.anaconda.org/gurobi`
- `conda install gurobi`

Warning

The Anaconda prompt application is NOT the same as the Command Prompt (Windows) or Terminal (MacOS)

If you don't have the Anaconda installation, then you can do something similar with the `pip` command, most likely, for those familiar with it.

2.3.6 Test your installation

Now you can test whether everything is setup correctly. Open an interactive Python session, for instance using Spyder. Try the following commands by adding them into a script (make sure there are no spaces at the beginning of the lines) and press Enter.

```python
from gurobipy import Model
model = Model()
```

```
Set parameter Username
Set parameter LicenseID to value 2715549
```

```
Academic license - for non-commercial use only - expires 2026-09-29
```

If both these commands succeed, you should see some output related to your academic license.

If the first command fails, then the Gurobi python module has not been installed correctly. If the second command fails, then the license has not been setup correctly (make sure the license file is at the right location).

If at any point, you need more information about Gurobi, then you can always go to the official Gurobi documention online or send the teacher an e-mail.

Chapter 3

Python basics

In this chapter we will learn how to use Python as a calculator and see some basic programming concepts.

3.1 Math operations

We start with the most basic mathematical (arithmetic) operations: Addition, subtraction, multiplication and division are given by the standard `+`, `-`, `*` and `/` operators that you would use in other programs like Excel. For example, addition:

```python
print(2 + 3)
```

```
5
```

Subtraction:

```python
print(5 - 3)
```

```
2
```

Multiplication:

```python
print(2 * 3)
```

```
6
```

Division:

```python
print(3 / 2)
```

```
1.5
```

It is also possible to do multiple operations at the same time using parentheses. For example, suppose we wanted to calculate:

$$\frac{2+4}{4\cdot 2} = \frac{6}{8} = 0.75$$

We can calculate this in Python as follows:

```
print((2 + 4) / (4 * 2))
```

```
0.75
```

With the `**` operator (two stars) we can raise a number to the power of another number. For example, $2^3 = 2 \times 2 \times 2 = 8$ can be computed as

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

```
8
```

⚠ WARNING

Do **not** use `^` for exponentiation. This actually does a very different thing in Python.

Exercise 3.1

Compute the following expressions using the operator $+, -, *, /$ and $**$:

i) $3 + 5 \cdot 2$

ii) $\frac{(10-4)^2}{3}$

iii) $\frac{((2+3)\cdot 4-5)^2}{3+1}$

3.2 Variables

In Python we can assign single numbers, as well as text, to *variables* and then work with and manipulate those variables.

Assigning a single number to a variable is very straightforward. We put the name we want to give to the variable on the left, then use the `=` symbol as the *assignment operator*, and put the number to the right of the `=`. The `=` operator binds a number (on the right-hand size of `=`) to a name (on the left-hand side of `=`).

To see this at work, let's set $x = 2$ and $y = 3$ and calculate $x + y$:

```
x = 2
y = 3
print(x + y)
```

```
5
```

When we assign $x = 2$, in our code, the number is not fixed forever. We can assign a new number to `x`. For example, we can assign the number 6 to `x` instead. The sum of x (which is 6) and y (which is 3), is now 9:

```
# Original variable assignment
x = 2
y = 3
print(x + y)
```

```
# Overwriting assignment of variable x
x = 6
print(x + y)
```

```
5
9
```

Finally, you cannot set $x = 2$ with the command `2 = x`. That will result in an error. The name must be on the left of `=` and the number must be on the right of `=`.

Exercise 3.2

Define variables a, b, c with numbers $19, 3$ and 7, respectively. Compute the following expressions:

i) $a + b \cdot c$
ii) $\frac{(a-c)^2}{b}$
iii) $\frac{((b+c)\cdot a-c^2)^2}{a+b}$

3.3 Lists

We can also store multiple variables in one object, a so-called *list*. A list with numbers is created by writing down a sequence of numbers, separated by commas, in between two brackets `[` and `]`.

```
z = [3, 9, 1, 7]
print(z)
```

```
[3, 9, 1, 7]
```

We can also create lists with fractional numbers.

```
z = [3.1, 9, 1.9, 7]
print(z)
```

```
[3.1, 9, 1.9, 7]
```

To access the numbers in the list, we can *index* the list at the position of interest. If we want to get the number at position i in the list, we use the syntax `z[i]`.

```
print(z[1])
```

```
9
```

Something strange is happening here… The left-most number in the list is 3.1, but `z[1]` returns 9. This happens because Python actually starts counting at index 0 (instead of 1).

Indexing convention in Python

The *left-most number* in a Python list is located at *position* 0. The number next to that at position 1, etc. That is, the i-th number in a list with n numbers can be found at position $i-1$ for $i = 1, \dots, n$

In other words, the “first” number in the list is located at position 0, and we can access it using `z[0]` instead. Below we index the number of the list at positions $i \in \{0, 1, 2, 3\}$ separately.

```
print(z[0])
```

```
3.1
```

```
print(z[1])
```

```
9
```

```
print(z[2])
```

```
1.9
```

```
print(z[3])
```

```
7
```

Exercise 3.3

Consider the list $a = [11, 41, 12, 35, 6, 33, 7]$.

i) Compute the sum of the numbers at even positions in a (i.e., positions $0, 2, 4$, and 6).
ii) Compute the result of multiplying the first and last elements of a, then subtracting the middle element.
iii) Compute the square of the element at position 2, divided by the sum of the elements at the odd positions.

3.4 Printing text

It is also possible to print text in Python with the `print()` command. To do this, you have to put the desired text in quotation marks, either single `''` or double `""`. It does not matter if you use single or double quotation as long as you use the same type on both end of the text.

```
print('Hello world!')
```

```
Hello world!
```

```
print("Hello world!")
```

```
Hello world!
```

A piece of text within quotation marks is called a *string* in Python. It is also possible to store text (strings) in a list, and print it from the list. For example, we can store the days of the weeks as text in the list `days` as follows.

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]
```

If we now want to print a day, we should index the list at the corresponding position. For example, to print `"Wednesday"`, we should index the list at position 2.

```
print(days[2])
```

```
Wednesday
```

If you want to print both text(s) and variables within the same `print()` command, you can do this by separating these quantities with a comma.

```
print("The third day of the week is", days[2])
```

```
The third day of the week is Wednesday
```

Exercise 3.4

Create a list called `courses` that contains five course names as text/strings. Then:

i. Print the first course in the list.
ii. Print the last course in the list.
iii. Print a sentence that says: My favorite color is where should be your favorite course indexed from the list.

3.5 For-loop

For-loops are a convenient way to avoid unnecessary repetition of different lines of code. Suppose we want to print all the days of the week from the `days` list.

We can do this by using a print statement for every day separately by indexing `days` at all its positions.

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

print("The days of the week are:")
print(days[0])
print(days[1])
print(days[2])
print(days[3])
print(days[4])
print(days[5])
print(days[6])
```

```
The days of the week are:
Monday
Tuesday
Wednesday
Thursday
```

```
Friday
Saturday
Sunday
```

However, looking at the above, the only thing that changes in the last seven lines of code is the position at which we access `days`. We can also print all the days of the week based on the list `days` above using a for-loop.

```python
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in range(0,7):
    print(days[i])
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

With the line `for i in range(0,7):` we tell Python that we would like to carry out a piece of code (the line below it) with different values of the *iteration variable* `i` ranging from 0 to 7, but NOT including 7. In other words, we want to run a piece of code for the values of `i` being $0, 1, 2, 3, 4, 5$ and 6. It is important to realize here that the index 7 is not included as value for `i` in the for-loop (this is what the Python developers decided on).

In Python such a range, called the *iterable*, can be specified with the `range(a,b)` function where a is the smallest value and $b - 1$ the largest value of i. In the first *iteration* with $i = 0$ Python will now print `days[0]`, in the second iteration with $i = 1$ Python will print `days[1]`, etcetera.

The code we want to execute repeatedly can be found under the lines `for i in range(0,7):` with an *indentation (or tab)*. The indendation is important so that Python known this is the line of code you want to be carried out repeatedly. We can also carry out multiple lines of code in a for-loop, then all these lines should be indented.

```python
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in range(0,7):
    print("Day", i+1, "of the week is:")
    print(days[i])
```

```
Day 1 of the week is:
Monday
Day 2 of the week is:
Tuesday
Day 3 of the week is:
Wednesday
Day 4 of the week is:
```

```
Thursday
Day 5 of the week is:
Friday
Day 6 of the week is:
Saturday
Day 7 of the week is:
Sunday
```

Note that in the first line in the for-loop, here we print three aspects in one `print()` statement: The word `Day`, the value of `i+1`, and the words `of the week is:`. As i ranges from 0 to 6, the values of $i + 1$ ranges from 1 to 7.

We note that we only use two print statements for the sake of illustration. The above could have been printed with one `print()` statement as well.

```python
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in range(0,7):
    print("Day", i+1, "of the week is:", days[i])
```

```
Day 1 of the week is: Monday
Day 2 of the week is: Tuesday
Day 3 of the week is: Wednesday
Day 4 of the week is: Thursday
Day 5 of the week is: Friday
Day 6 of the week is: Saturday
Day 7 of the week is: Sunday
```

Exercise 3.5

Take your list `courses` from the previous exercise and consider the list `days` from above. Print the five sentences: “On Monday the course [course0] is being taught”, “On Tuesday the course [course1] is being taught”, etc. by using a for-loop in which you index the courses and days from the two lists, and where [course0], [course1], etc. are the first, second, etc. course names from the list.

We remark that it is also possible to directly iterate over the values in a list (the iterable).

```python
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in days:
    print(i)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

```
Sunday
```

In this case, i does not range from 0 to 6, but in iterates directly over the strings in the list `days`. So in the first iteration $i =$ "Monday", in the second iteration $i =$ "Tuesday", etcetera.

3.6 Conditional statements

It is also possible in Python have different lines of code executed based on conditions. For this we use an **if/else statement**. This allows the program to execute certain lines of code only when a condition is true.

For example, suppose we store a number in the variable x and want to print a message only if x is greater than 5.

```
x = 8

if x > 5:
    print("x is greater than 5")
```

```
x is greater than 5
```

The line if `x > 5:` checks whether the condition x > 5 is true. If it is true, Python executes the indented line below it. If the condition is false, nothing happens and Python continues with the rest of the code. You can check this yourself by changing the value of x to something strictly smaller than 5. If you run the code snippet, nothing should be printed.

We can also tell Python what to do if the condition is false by adding an **else statement**.

```
x = 3

if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

```
x is not greater than 5
```

Here Python checks the condition. Since 3 > 5 is false, i.e., the condition $x > 5$ is not true, the code under `else:` is executed instead.

Just like with for-loops, **indentation is important**. All lines that belong to the if or else block must be indented so Python knows which code belongs to which condition.

We can also check more than two conditions using `elif` (short for "else if"):

```
x = 5

if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is exactly 5")
    print("You can execute multiple lines of code when a condition is true")
else:
    print("x is less than 5")
```

```
x is exactly 5
You can execute multiple lines of code when a condition is true
```

In this case Python checks the conditions from top to bottom and executes only the first one that is true. In the snippet above the line `"x is exactly 5"` will be printed. You can play around with this code snippet by changing the values of x and observing that, depending on its value, a different line will be printed.

You can use the following symbols to compare a variable with a given number.

Operator	Meaning	Example (`x = 5`)	Result
`<`	Less than	`x < 7`	`True`
`>`	Greater than	`x > 7`	`False`
`<=`	Less than or equal to	`x <= 5`	`True`
`>=`	Greater than or equal to	`x >= 6`	`False`
`==`	Equal to	`x == 5`	`True`
`!=`	Not equal to	`x != 5`	`False`

You can also combine multiple statements into one condition using the `and` and `or` *keyword arguments*. In the code below, we do not finish with an `else` statement; this is not mandatory.

```
x = 5

if x > 5 and x < 10:
    print("x is between 5 and 10")
elif x == 5 or x == -5:
    print("The absolute value of x is 5")
```

```
The absolute value of x is 5
```

You can also combine for-loops and if/else statements. For example, recall the original if/else statement where we checked whether x was greater or equal than 5.

```
x = 3

if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

```
x is not greater than 5
```

We can also check this for a collection of numbers stored in a list `numbers` by iterating with a for-loop over these numbers and check the conditions for every element in the list.

```python
numbers = [3, 7, -3, 4, 10]

for x in numbers:
    if x > 5:
        print("x is greater than 5")
    else:
        print("x is not greater than 5")
```

```
x is not greater than 5
x is greater than 5
x is not greater than 5
x is not greater than 5
x is greater than 5
```

Note that in the above code we use indentation twice! For every iteration of the for-loop we want to check the if/else statement, so all those lines need to be indented. Because it also needs to be clear, for every iteration, which lines correspond to the if-statement, and which to the else-statement, the `print()` statements are indented once more.

Exercise 3.6

Consider the list of temparatures (in degrees Celsius) $[20, 14, 12, 18, 25, 30, 31]$ corresponding to the days of the week Monday through Sunday. For every day you have to print the message "[day] is a [cold/normal/hot] summer day", where a day is cold if the temperature is below 15, normal if it is in the interval $[15, 25]$ (i.e., greater or equal than 15 and smaller or equal than 25), and hot if it is higher than 25.
The output of your code should be:

```
Monday is a normal summer day.
Tuesday is a cold day.
Wednesday is a cold day.
Thursday is a normal summer day.
Friday is a normal summer day.
Saturday is a hot summer day.
Sunday is a hot summer day.
```

Chapter 4

Linear optimization

In this chapter we will show how to implement linear optimization problems using Python. The goal is to model optimization problems in Python and then solve them using Gurobi, which is a software package dedicated to solving optimization problems. Note that it is important that you installed Gurobi correctly as explained in Chapter 2.

4.1 Basics

All the relevant functionality needed to use Gurobi via Python is contained in the `gurobipy` package. For our purposes, we only need two *modules* from this package:

- The `Model` module that we will use to build optimization problems
- The `GRB` module that contains various "constants" that we use to include words such as *maximize* and *minimize*.

In Python you can include the modules by adding the following line at the top of a script.

```python
from gurobipy import Model, GRB
```

Example: Duplo problem

To illustrate how to implement optimization problems in Python, we consider the Duplo problem from the lectures:

$$\begin{array}{lll} \text{maximize} & 15x_1 + 20x_2 & \text{(profit)} \\ \text{subject to} & x_1 + 2x_2 \leq 6 & \text{(big bricks)} \\ & 2x_1 + 2x_2 \leq 8 & \text{(small bricks)} \\ & x_1, x_2 \geq 0 & \end{array}$$

with decision variables

- x_1: Number of chairs
- x_2: Number of tables

Let us implement the Duplo problem in Gurobi.

4.1.1 Model() object

To start, we will create a variable (or object) `duplo` that will contain all the information of the problem, such as the decision variables, objective function and constraints.

In programming terms, we create an *object* from the `Model()` class. We can give the instance a name by adding the `name` *keyword argument*, with value `'Duplo problem'` in our case. We add this name with quotations so that Python knows this is plain text.

```
# Initialize Gurobi model.
duplo = Model(name='Duplo problem')
```

```
Set parameter Username
Set parameter LicenseID to value 2715549
Academic license - for non-commercial use only - expires 2026-09-29
```

You can think of the Model class object `duplo` as a large box that we are going to fill with decision variables, an objective function, and constraints. We will also reserve a small space in this box to store information about the optimal solution to the linear optimization problem, once we have optimized the problem.

The concept of having “objects” is in fact what the programming language Python is centered around. This is known as *object-oriented programming*.

4.1.2 Decision variables

To add decision variables, an objective function, and constraints, to our Model object we use so-called *methods* which are Python functions.

To create a decision variable, we can use the `addVar()` method. As input argument, we include a name for the decision variables.

```
# Declare the two decision variables.
x1 = duplo.addVar(name='chairs')
x2 = duplo.addVar(name='tables')
```

> If your Model object is not called `duplo`, but for example `production_planning`, then you should use `production_planning.addVar(name='chairs')` instead. The same applies for all later methods that are introduced. Never call a model `Model` (with capital M) because this spelling is reserved for the `Model()` class in Gurobi.

4.1.3 Objective function

Now that we have our decision variables, we can use them to define the objective function and the constraints.

We use the `setObjective()` method to do so by refering to the variables created above. The `sense` keyword argument must be used to specify whether we want to maximize or minimize the objective function. For this, we use `GRB.MAXIMIZE` or `GRB.MINIMIZE`, respectively.

```
# Specify the objective function.
duplo.setObjective(15*x1 + 20*x2, sense=GRB.MAXIMIZE)
```

4.1.4 Constraints

The next step is to declare the constraints using the `addConstr()` method. Also here, we can specify the constraints refering to the variables `x1` and `x2`. We give the constraints a name as well.

```
# Add the resource constraints on bricks.
duplo.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
duplo.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')
```

```
<gurobi.Constr *Awaiting Model Update*>
```

We can also add the nonnegativity constraints $x_1 \geq 0$ and $x_2 \geq 0$ in this way, but this is actually not needed! By default, Python adds in the background the constraints that the decision variables should be nonnegative when creating them (see also Section 4.1.6).

4.1.5 Optimal solution

Now the model is completely specified, we are ready to compute the optimal solution. We can do this using the `optimize()` method applied to our model `duplo`.

```
# Optimize the model
duplo.optimize()
```

```
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 10.0 (19045.2))

CPU model: 12th Gen Intel(R) Core(TM) i7-1265U, instruction set [SSE2|AVX|AVX2]
Thread count: 10 physical cores, 12 logical processors, using up to 12 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xadc88607
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [2e+01, 2e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 8e+00]
Presolve time: 0.00s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    3.5000000e+31   3.500000e+30   3.500000e+01      0s
       2    7.0000000e+01   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.01 seconds (0.00 work units)
Optimal objective  7.000000000e+01
```

We can see some output from Gurobi, and the last line tells us that Gurobi found an optimal solution with objective value 70.

To summarize all the step above, we have given the complete code below (but not executed this time).

```python
from gurobipy import Model, GRB

# Initialize Gurobi model.
duplo = Model(name='Duplo problem')

# Declare the two decision variables.
x1 = duplo.addVar(name='chairs')
x2 = duplo.addVar(name='tables')

# Specify the objective function.
duplo.setObjective(15*x1 + 20*x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
duplo.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
duplo.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')

# Optimize the duplo
duplo.optimize()
```

Recalling our linear optimization problem object as being a "box" filled with decision variables, an objective function and constraints, the box now also contains a small space where the optimal solution is stored.

Let us have a look at how to access information about the optimal solution. The objective function value of the optimal solution, that was also displayed in the output when optimization the model with `duplo.optimize()` is stored in the `ObjVal` *attribute*. An attribute is a piece of information about an object in Python.

```python
# Print text 'Objective value:' and the duplo.ObjVal attribute value
print('Objective value:', duplo.ObjVal)
```

```
Objective value: 70.0
```

The optimal values of a decision variables can be obtained by accessing the `X` *attribute* of a variable.

```python
print('x1 = ', x1.X)
print('x2 = ', x2.X)
```

```
x1 =  2.0
x2 =  2.0
```

If the model has many variables, printing variables in this way is a cumbersome approach. It is then easier to iterate over all variables of the model using the `getVars()` method. This method creates a list of all the decision variables of the model.

```python
print(duplo.getVars())
```

```
[<gurobi.Var chairs (value 2.0)>, <gurobi.Var tables (value 2.0)>]
```

As you can see above, we already see the optimal values of the decision variables, but also a lot of unnecessary information.

Let us print these values in a nicer format by iterating over the list `duplo.getVars()` using a for-loop with index variable `var`; you can choose another name here if you want. We print for every variable `var` its name, stored in the attribute `VarName`, and its value, stored in the attribute `X`. In between, we print the = symbol in plain text (hence, the quotations).

Recall that if you want to print multiple date types (such as a variables and text) in a `print()` statement, you should separate them with commas.

```python
for var in duplo.getVars():
    print(var.VarName, "=", var.X)
```

```
chairs = 2.0
tables = 2.0
```

4.1.6 Help functionality

To see all these properties of method, e.g., `addVar`, you can check out the `addVar` documentation by typing `help(Model.addVar)` in Python. Looking up the documentation in this way can be done for every method.

```python
help(Model.addVar)
```

```
Help on cython_function_or_method in module gurobipy._model:

addVar(self, lb=0.0, ub=1e+100, obj=0.0, vtype='C', name='', column=None)
    ROUTINE:
      addVar(lb, ub, obj, vtype, name, column)

    PURPOSE:
      Add a variable to the model.

    ARGUMENTS:
      lb (float): Lower bound (default is zero)
      ub (float): Upper bound (default is infinite)
      obj (float): Objective coefficient (default is zero)
      vtype (string): Variable type (default is GRB.CONTINUOUS)
      name (string): Variable name (default is no name)
      column (Column): Initial coefficients for column (default is None)

    RETURN VALUE:
      The created Var object.

    EXAMPLE:
      v = model.addVar(ub=2.0, name="NewVar")
```

For example, from this it can be seen that when creating a variable, you can specify more properties. You can use the keyword arguments `lb` and `ub` to define a lower and upper bound, respectively, for a decision variable.

If we would want to create a variable, with name *Test variable*, having lower bound 3 and upper bound 7, i.e., these bounds model the constraints $x_1 \geq 3$ and $x_1 \leq 7$, we can do this with `x = duplo.addVar(lb=3, ub=7 ,name='Test variable')` using the `lb` and `ub` keyword arguments.

As you can see above, all the keyword arguments have default values, meaning that if we do not specify them, Gurobi uses the specified default value for them (and sets these outside of our view). For example, by default a variable is continuous (`vtype` keyword argument) and non-negative (`lb` keyword argument). This is the reason why we do not have to specify the nonnegativity constraints $x_1, x_2 \geq 0$ for the Duplo example.

Also note that `ub` is set to 10^{100} which roughly speaking indicates that the decision variable has no upper bound value by default (because it is highly unlikely that an optimal solution would have a value that large).

4.2 Sensitivity analysis

In class we have investigated how the optimal solution changes when either an objective function coefficient changes, or if one of the right hand side values of a constraint changes.

To obtain a concise report with all the OFC ranges of the objective function coefficients, as well as the shadow prices and allowable ranges of all the constraints, right-click and store the Python file (sensitivity_analysis.py) in the same folder as where you stored your linear optimization problem.

Make sure the name of the file you download here is *sensitivity_analysis* and that it is stored as a .py file. On Windows you would use “Save as type: All files” and then store the file as *sensitivity_analysis.py*.

In the Python file containing your linear optimization problem, you must include the lines below where you should replace `[model_object]` by the variable name of your model. For the Duplo example, this was `duplo`.

```
from sensitivity_analysis import report
report([model_object])
```

The following code defines again the Duplo problem in Gurobi and generates the sensitivity report.

```
from gurobipy import Model, GRB

# Initialize Gurobi model.
duplo = Model(name='Duplo problem')

# Declare the two decision variables.
x1 = duplo.addVar(name='chairs')
x2 = duplo.addVar(name='tables')

# Specify the objective function.
duplo.setObjective(15*x1 + 20*x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
```

```python
duplo.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
duplo.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')

# Optimize the duplo
duplo.optimize()

# Generate sensitivity report
from sensitivity_analysis import report
report(duplo)
```

```
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 10.0 (19045.2))

CPU model: 12th Gen Intel(R) Core(TM) i7-1265U, instruction set [SSE2|AVX|AVX2]
Thread count: 10 physical cores, 12 logical processors, using up to 12 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xadc88607
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [2e+01, 2e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 8e+00]
Presolve time: 0.00s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    3.5000000e+31   3.500000e+30   3.500000e+01      0s
       2    7.0000000e+01   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.00 seconds (0.00 work units)
Optimal objective  7.000000000e+01
```

Sensitivity Report

Variables	Opt. Value	Obj. Coeff.	OFC Range
chairs	2.0	15.0	[10.0, 20.0]
tables	2.0	20.0	[15.0, 30.0]

Constraints	Shadow Price	RHS	Allowable Range
big-bricks	5.0	6.0	[4.0, 8.0]
small-bricks	5.0	8.0	[6.0, 12.0]

4.3 Beyond the basics

When building optimization models in gurobipy, you often need many decision variables and long sums in objectives or constraints. Instead of writing everything manually, Gurobi provides tools to do this compactly and clearly.

Two especially useful tools are:

- `addVars()` : Create many variables at once
- `quicksum()` : Efficiently create constraints in terms of decision variables

4.3.1 Defining many variables

If we define a `Model()` object with name `problem`, we can use the syntax `x = problem.addVars(n)` to tell Python to create n decision variables. You can use the i-th decision variable to define constraints and the objective function by indexing `x` at position $i-1$, i.e, by using `x[i-1]` . Recall that Python starts counting from zero when indexing data objects such as lists.

If you want you can also add a list of names for the decision variables using the `name` keyword argument. Make sure that the number of elements in the list matches the value of n.

Below we use `addVars()` to again define the Duplo example with $n = 2$.

```
from gurobipy import Model, GRB

# Initialize Gurobi model.
```

```
duplo = Model(name='Duplo problem')

# Declare the two decision variables using addVars()
x = duplo.addVars(2,name=["chairs","tables"])

# Specify the objective function in terms of x[0] and x[1]
duplo.setObjective(15*x[0] + 20*x[1], sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks in terms of x[0] and x[1]
duplo.addConstr(x[0] + 2*x[1] <= 6, name='big-bricks')
duplo.addConstr(2*x[0] + 2*x[1] <= 8, name='small-bricks')

# Optimize the duplo
duplo.optimize()
```

4.3.2 Defining constraints quickly

The function `quicksum()` is convenient to use if you you want to add the objective function $c_1x_1 + x_2x_2 + \cdots + c_nx_n$ or a constraint of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$. You have to import this function into Python just as we have to do with `Model` and `GRB`.

Let us again consider the Duplo example. If we define $x = [x_1, x_2]$ and $c = [c_1, c_2] = [2, 2]$, then we can create the objective function $c_1x_1 + c_2x_2$ by first creating the list $[c_1x_1, c_2x_2]$, containing the individual terms of the objective function, and then summing up the elements in this list. The latter is precisely what `quicksum()` does: It takes as input a list and adds up the number in this list.

In the code below we do two things:

1. Create the list `y = [c[0]*x[0], c[1]*x[1]]` using the concept of *list comprehension*.
2. Use the `quicksum()` function to add up the elements in the list `y` and set the resulting variable `obj` as our objective function.

```
from gurobipy import Model, GRB, quicksum

# Initialize Gurobi model.
duplo = Model(name='Duplo problem')

# Declare the two decision variables using addVars()
x = duplo.addVars(2,name=["chairs","tables"])

# Coefficients of objective function
c = [2,2]

# Define list [c[0]*x[0], c[1]*x[1]]
y = [c[j]*x[j] for j in range(2)]

# Sum up the terms in the list y
```

```
obj = quicksum(y)

# Define objective
duplo.setObjective(obj, sense=GRB.MAXIMIZE)
```

The part `[c[j]*x[j] for j in range(2)]` is the compact *list comprehension* syntax for creating `[c[0]*x[0], c[1]*x[1]]`. Recall from Section 3.5 that `j in range(0,2)` means we iterate over the values $j = 0$ and $j = 1$ (the value 2 is not included).

This time we are only creating a list with two elements, but imagine then if you have 100 variables, then `[c[j]*x[j] for j in range(100)]` would be all you need to create an objective function of the form $c_1x_1 + c_2x_2 + \cdots + c_{100}x_{100}$ which consists of 100 terms!

We remark that you can also combine the above steps into a more compact code that carries out the last three lines in one line.

```
# Define objective
duplo.setObjective(quicksum(c[j]*x[j] for j in range(2)), sense=GRB.MAXIMIZE)
```

In a similar fashion you can define the left hand side of a constraint $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$ quickly with `quicksum()`. Below we have given the Duplo example implemented with the use of `quicksum()` and `addVars()`.

```
from gurobipy import Model, GRB

# Initialize Gurobi model.
duplo = Model(name='Duplo problem')

# Declare the two decision variables using addVars()
x = duplo.addVars(2,name=["chairs","tables"])

# Objective function coefficients
c = [2,2]

# Big brick coefficients
b = [1,2]

# Small brick coefficients
s = [2,2]

# Specify the objective function in terms of x[0] and x[1]
duplo.setObjective(quicksum(c[j]*x[j] for j in range(2)), sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks in terms of x[0] and x[1]
duplo.addConstr(quicksum(b[j]*x[j] for j in range(2)) <= 6, name='big-bricks')
duplo.addConstr(quicksum(s[j]*x[j] for j in range(2)) <= 8, name='small-bricks')
```

```
# Optimize the duplo
duplo.optimize()

# Print optimal values of decision variables
for var in duplo.getVars():
    print(var.VarName, "=", var.X)
```

```
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 10.0 (19045.2))

CPU model: 12th Gen Intel(R) Core(TM) i7-1265U, instruction set [SSE2|AVX|AVX2]
Thread count: 10 physical cores, 12 logical processors, using up to 12 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0x1b7dc040
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [2e+00, 2e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 8e+00]
Presolve time: 0.00s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    4.0000000e+30   3.500000e+30   4.000000e+00      0s
       2    8.0000000e+00   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.00 seconds (0.00 work units)
Optimal objective  8.000000000e+00
chairs = 2.0
tables = 2.0
```

4.4 Integer variables

If you are not interested in finding fractional solutions, but want to enforce the decision variables to take integer values, you can do this with the `vtype` keyword argument.

Consider the following linear optimization problem (without integrality constraints).

$$\begin{array}{llrcrcr} \max & z = & 2x_1 & + & x_2 & & \\ \text{s.t.} & & x_1 & + & x_2 & \leq & 2.5 \\ & & x_1, & & x_2 & \geq & 0 \end{array},$$

```
# Initialize Gurobi model.
model_frac = Model(name='Fractional problem')

# Suppress Gurobi Optimizer output
```

```python
model_frac.setParam('OutputFlag',0)

# Declare the two decision variables.
x1 = model_frac.addVar(name="x1")
x2 = model_frac.addVar(name="x2")

# Specify the objective function.
model_frac.setObjective(2*x1 + x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
model_frac.addConstr(x1 + x2 <= 2.5)

# Optimize the model
model_frac.optimize()

# Print optimal values of decision variables
for var in model_frac.getVars():
    print(var.VarName, "=", var.X)
```

```
x1 = 2.5
x2 = 0.0
```

If we specify `vtype=GRB.INTEGER` in the `addVar()` method when creating the variables, Gurobi will find an optimal solution with all decision variables being integers.

```python
# Initialize Gurobi model.
model_int = Model(name='Integer problem')

# Suppress Gurobi Optimizer output
model_int.setParam('OutputFlag',0)

# Declare the two decision variables.
x1 = model_int.addVar(name="x1",vtype=GRB.INTEGER)
x2 = model_int.addVar(name="x2",vtype=GRB.INTEGER)

# Specify the objective function.
model_int.setObjective(2*x1 + x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
model_int.addConstr(x1 + x2 <= 2.5)

# Optimize the model
model_int.optimize()

# Print optimal values of decision variables
for var in model_int.getVars():
    print(var.VarName, "=", var.X)
```

```
x1 = 2.0
x2 = -0.0
```

Anoter common case is where the decision variables are supposed to be binary, meaning they can only take values in $\{0, 1\}$. Setting variables to be binary can be done with `vtype=GRB.BINARY`.

```python
# Initialize Gurobi model.
model_bin = Model(name='Binary problem')

# Suppress Gurobi Optimizer output
model_bin.setParam('OutputFlag',0)

# Declare the two decision variables.
x1 = model_bin.addVar(name="x1",vtype=GRB.BINARY)
x2 = model_bin.addVar(name="x2",vtype=GRB.BINARY)

# Specify the objective function.
model_bin.setObjective(2*x1 + x2, sense=GRB.MAXIMIZE)

# Add the resource constraints on bricks.
model_bin.addConstr(x1 + x2 <= 2.5)

# Optimize the model
model_bin.optimize()

# Print optimal values of decision variables
for var in model_bin.getVars():
    print(var.VarName, "=", var.X)
```

```
x1 = 1.0
x2 = 1.0
```

Chapter 5

Monte Carlo simulation

In this chapter we show how to solve an optimization problem for many scenarios, where each scenario has slightly different input data. We also consider the generation of random scenarios.

5.1 Scenario analysis

Consider again the Duplo example with objective function coefficients s_1 and s_2:

$$\begin{array}{lll} \text{maximize} & s_1 \cdot x_1 + s_2 \cdot x_2 & \text{(profit)} \\ \text{subject to} & x_1 + 2x_2 \leq 6 & \text{(big bricks)} \\ & 2x_1 + 2x_2 \leq 8 & \text{(small bricks)} \\ & x_1, x_2 \geq 0 & \end{array}$$

with decision variables

- x_1: Number of chairs
- x_2: Number of tables

We can solve the Duplo example for different objective function coefficients, with the results as specified in the table below.

Scenario	s_1	s_2	**Optimal solution**
1	15	20	(2,2)
2	9	25	(0,3)
3	20	19	(4,0)
4	16	21	(2,2)

To solve the Duplo problem repeatedly in Python using a different scenario every time, we can use a for-loop. We specify the different values of s_1 and s_2 in two separate lists, where `s1[i]` is the value of s_1 in scenario $i+1$ (recall Python starts indexing at zero), and similarly `s2[i]` the value of s_2 in scenario $i+1$.

```
from gurobipy import Model, GRB

# Different scenarios
```

```python
s1 = [15, 9, 20, 16]
s2 = [20, 25, 19, 21]

for i in range(0,4):
    # Initialize Gurobi model.
    duplo = Model(name='Duplo problem')

    # Suppress Gurobi Optimizer output
    duplo.setParam('OutputFlag',0)

    # Declare the two decision variables.
    x1 = duplo.addVar(name='chairs')
    x2 = duplo.addVar(name='tables')

    # Specify the objective function in terms of s1[i], s2[i]
    duplo.setObjective(s1[i]*x1 + s2[i]*x2, sense=GRB.MAXIMIZE)

    # Add the resource constraints on bricks.
    duplo.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
    duplo.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')

    # Optimize the duplo
    duplo.optimize()

    # Print solution
    print("Optimal solution for scenario", i+1, "is [", x1.X, ",", x2.X, "]")
```

```
Set parameter Username
Set parameter LicenseID to value 2715549
Academic license - for non-commercial use only - expires 2026-09-29
Optimal solution for scenario 1 is [ 2.0 , 2.0 ]
Optimal solution for scenario 2 is [ 0.0 , 3.0 ]
Optimal solution for scenario 3 is [ 4.0 , 0.0 ]
Optimal solution for scenario 4 is [ 2.0 , 2.0 ]
```

A couple of remarks are in place for the code above. First of all, note that all the lines below the line `for i in range(0,4):` are indented because we want the model to be constructed, optimized and printed in every iteration.

We use the command `duplo.setParam('OutputFlag',0)` to suppress the output that Gurobi usually generates when we optimize a linear optimization problem. We do this to avoid that this output is printed in every interation of the for-loop.

Also note that the objective function is defined as `s1[i]*x1 + s2[i]*x2` meaning that in iteration i, the objective function uses the values `s1[i]` and `s2[i]` (which are therefore different in every iteration as they depend on i).

In the `print()` statement we print a combination of text and the values `i+1`, `x1.X` and `x2.X` where the latter two are the values of the optimal solution in scenario $i + 1$.

5.2 Random scenarios

In the previous section, we chose the values of the objective function coefficients in every scenario ourselves. These can also be generated randomly using random number generation functions in Python contained in the *NumPy package*, which contains a lot of the mathematical functionality that Python has to offer.

This package is typically imported under the *alias* `np`, which is shorthand notation so that we do not have to type `numpy` everytime.

```
import numpy as np
```

We treat two functions for generating random numbers. The first one can be used to generate fractional random numbers.

- `np.random.rand(n)` : Generates n random numbers from the interval $[0, 1]$.

The generated numbers are stored in a list. Technically speaking, they are stored in a NumPy array, which you can think of as a "mathematical" list (but this is not important for us).

```
import numpy as np

# Generate five random (fractional) numbers from [0,1]
x = np.random.rand(5)

# Print the numbers
print(x)
```

```
[0.11080666 0.92005319 0.67696833 0.32282672 0.58709053]
```

Note that if you want to generate a (fractional) number from an interval $[a, b]$, you can do this by generating a number x from the interval $[0, 1]$, multiplying this number by $(b - a)$ and then adding a to the result.

In other words, if x is a random number from $[0, 1]$, then the number $a + (b - a) \cdot x$ is a random number from $[a, b]$.

```
import numpy as np

# Interval values
a = 3
b = 7

# Generate random (fractional) number from [0,1]
x = np.random.rand(1)

# Random number from interval [a,b] = [3,7]
y = a + (b-a)*x

# Random number from interval [a,b]
print(y)
```

```
[6.33654996]
```

Note that, in the above code snippet, even though we only generate one number, it is still given to us in a list. If you only want to get the number (not in a list) you should index the list at position 0. This is important if you want to use such a number for defining the objective function or a constraint.

```
print(y[0])
```

```
6.336549964224767
```

You can also generate multiple numbers from an interval $[a, b]$ if $x = [x_0, x_1, \dots]$ is a list of numbers generated from $[0, 1]$. If you multiply x with $(b - a)$, then all numbers in the list are multiplied by $(b - a)$, i.e., $(b - a) * x = [(b - a) * x_0, (b - a) * x_1, \dots]$. Similarly, if we add a to $(b - a) * x$, then a is added to every number in the list $(b - a) * x$, i.e., we get $a + (b - a) * x = [a + (b - a) * x_0, a + (b - a) * x_1, \dots]$.

```
import numpy as np

# Interval values
a = 3
b = 7

# Generate five random (fractional) numbers from [0,1]
x = np.random.rand(5)

# Five random numbers from interval [a,b] = [3,7]
y = a + (b-a)*x

# Print the numbers
print(y)
```

```
[4.74668105 3.88944326 3.96044803 4.59672441 5.87385957]
```

The second function we discuss can generate integer-valued numbers from a specified range of integers:

- `np.random.randint(low=a, high=b, size=n)` : Generates n random integers from the set $\{a, a + 1, \dots, b - 1\}$.

```
import numpy as np

# Generate five random integers from the set {2,3,4,5,6,7,8,9}
y = np.random.randint(low=2, high=10, size=5)

# Print the numbers
print(y)
```

```
[8 4 8 4 9]
```

We can use these random numbers to generate random scenarios and then solve the Duplo example for those scenarios.

```python
from gurobipy import Model, GRB

# Different 20 random scenarios
s1 = np.random.randint(low=15, high=25, size=20) # Numbers from {15,16,...,24}
s2 = np.random.randint(low=20, high=35, size=20) # Numbers from {20,12,...,34}

for i in range(0,20):
    # Initialize Gurobi model.
    duplo = Model(name='Duplo problem')

    # Suppress Gurobi Optimizer output
    duplo.setParam('OutputFlag',0)

    # Declare the two decision variables.
    x1 = duplo.addVar(name='chairs')
    x2 = duplo.addVar(name='tables')

    # Specify the objective function in terms of s1[i], s2[i]
    duplo.setObjective(s1[i]*x1 + s2[i]*x2, sense=GRB.MAXIMIZE)

    # Add the resource constraints on bricks.
    duplo.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
    duplo.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')

    # Optimize the duplo
    duplo.optimize()

    # Print solution
    print("Optimal solution for scenario", i+1, "is [", x1.X, ",", x2.X, "]")
```

```
Optimal solution for scenario 1 is [ 2.0 , 2.0 ]
Optimal solution for scenario 2 is [ 2.0 , 2.0 ]
Optimal solution for scenario 3 is [ 2.0 , 2.0 ]
Optimal solution for scenario 4 is [ 2.0 , 2.0 ]
Optimal solution for scenario 5 is [ 2.0 , 2.0 ]
Optimal solution for scenario 6 is [ 2.0 , 2.0 ]
Optimal solution for scenario 7 is [ 2.0 , 2.0 ]
Optimal solution for scenario 8 is [ 2.0 , 2.0 ]
Optimal solution for scenario 9 is [ 2.0 , 2.0 ]
Optimal solution for scenario 10 is [ 2.0 , 2.0 ]
Optimal solution for scenario 11 is [ 2.0 , 2.0 ]
Optimal solution for scenario 12 is [ 0.0 , 3.0 ]
Optimal solution for scenario 13 is [ 2.0 , 2.0 ]
Optimal solution for scenario 14 is [ 4.0 , 0.0 ]
Optimal solution for scenario 15 is [ 2.0 , 2.0 ]
Optimal solution for scenario 16 is [ 2.0 , 2.0 ]
Optimal solution for scenario 17 is [ 2.0 , 2.0 ]
Optimal solution for scenario 18 is [ 2.0 , 2.0 ]
Optimal solution for scenario 19 is [ 2.0 , 2.0 ]
Optimal solution for scenario 20 is [ 2.0 , 2.0 ]
```

Note that the solution $(2, 2)$ appears very often, meaning it is a stable solution in the sense that even though the objective function coefficients change, the solution remains optimal.

You can also generate the random numbers inside of the for-loop. Note that in this case you should use `s1[0]` and `s2[0]` in the objective function. As discussed before, this is because `s1 = np.random.randint(low=15, high=25, size=1)` is a list of length 1, and so to get the number in this list, you have to index `s1` at index 0.

Note that in every iteration, first two random numbers `s1` and `s2` are generated, which are then later used to define the objective function. Because new numbers are generated at the start of every iteration, we are solving a different problem every time.

```python
from gurobipy import Model, GRB

for i in range(0,20):
    # Generate random coefficients
    s1 = np.random.randint(low=15, high=25, size=1)
    s2 = np.random.randint(low=15, high=25, size=1)

    # Initialize Gurobi model.
    duplo = Model(name='Duplo problem')

    # Suppress Gurobi Optimizer output
    duplo.setParam('OutputFlag',0)

    # Declare the two decision variables.
    x1 = duplo.addVar(name='chairs')
    x2 = duplo.addVar(name='tables')

    # Specify the objective function in terms of s1[0], s2[0]
    duplo.setObjective(s1[0]*x1 + s2[0]*x2, sense=GRB.MAXIMIZE)

    # Add the resource constraints on bricks.
    duplo.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
    duplo.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')

    # Optimize the duplo
    duplo.optimize()

    # Print solution
    print("Optimal solution for scenario", i+1, "is [", x1.X, ",", x2.X, "]")
```

```
Optimal solution for scenario 1 is [ 4.0 , 0.0 ]
Optimal solution for scenario 2 is [ 2.0 , 2.0 ]
Optimal solution for scenario 3 is [ 4.0 , 0.0 ]
Optimal solution for scenario 4 is [ 2.0 , 2.0 ]
Optimal solution for scenario 5 is [ 4.0 , 0.0 ]
Optimal solution for scenario 6 is [ 4.0 , 0.0 ]
Optimal solution for scenario 7 is [ 2.0 , 2.0 ]
Optimal solution for scenario 8 is [ 2.0 , 2.0 ]
Optimal solution for scenario 9 is [ 4.0 , 0.0 ]
Optimal solution for scenario 10 is [ 2.0 , 2.0 ]
Optimal solution for scenario 11 is [ 4.0 , 0.0 ]
Optimal solution for scenario 12 is [ 4.0 , 0.0 ]
Optimal solution for scenario 13 is [ 2.0 , 2.0 ]
Optimal solution for scenario 14 is [ 2.0 , 2.0 ]
Optimal solution for scenario 15 is [ 2.0 , 2.0 ]
Optimal solution for scenario 16 is [ 4.0 , 0.0 ]
Optimal solution for scenario 17 is [ 2.0 , 2.0 ]
Optimal solution for scenario 18 is [ 2.0 , 2.0 ]
Optimal solution for scenario 19 is [ 4.0 , 0.0 ]
Optimal solution for scenario 20 is [ 4.0 , 0.0 ]
```