PYTHON
for
Civil and
Structural
Engineers

Vittorio Lora

Table of Contents

Part 1

Part 2

Introduction

About this book

You will find in this book is exactly what is written in the title: you will learn to use python to solve civil/mechanical related engineering problems. This might seem difficult at first, especially if you have never used python before, but really there is no need to worry. I have seen *plenty* of people (my colleagues, mostly) start from zero and get to the point where they use python every day in very little time. Once you start learning, you will realize how much potential this language has, and where it can sit in your everyday workflow. Personally, since I have started using python I have stopped using many of the programs that where my everyday go-to. Why? because with python I can do everything *much quicker*, and in a much more efficient way. For example, I can do all my calculations and typeset them *automatically* in a pdf document, complete with images and tables, ready for print. This means that there is non need to write all the results by hand, which saves time and leaves less room for errors to seep in.

What you will learn by reading this book is very practical knowledge, and that is why I have decided to structure the learning experience as a series of real world examples. I wanted to keep everything as closely related to engineering as possible, so apart from the first chapters where I explain the basics of the language, what follows is a compendium of code snippets that you will certainly find useful in your work environment. But what are the examples about? I come from a structural civil engineering background, and this book is targeted mostly toward people who have studied (or are studying) civil or mechanical engineering. There are examples about member design, forces diagrams plotting, section verification, finite element analysis, and much more. I hope you will follow trough all of them, and apply your newly acquired skills in real life.

The source code of every example in this book can be downloaded from https://python4civil.weebly.com/book-resources.html

For whom is this book written for?

Broadly speaking, this book is targeted to anybody who wishes to learn a solid programming language and use it in civil and mechanical applications. However, even if you come from another field of study, you will find in here really useful cross-disciplinary techniques that you can apply in your specific field of expertise. If I had to pinpoint a group of people who I think would benefit greatly from this book, my choice would fall on those who rely heavily on Excel for their work. Everything you do in Excel can be done in python, and in a much more efficient way. If you have ever designed a concrete beam using Excel, then you know that the spreadsheets can become quite overcrowded in a very short time. Essentially, if you fall in one of these categories, then this book is for you:

- You want to transition from Excel to something faster and more flexible;
- You want to learn how to use python for practical engineering applications;
- You are a university student looking for a well-explained set of examples on designing structural elements;
- You want to use python in your work environment, but you still wish to share Excel documents and tables with your colleagues;
- You want to learn how to automatically create professionally formatted reports from your calculations, and export them in LaTeX or pdf format;
- you want to learn how to plot beautiful graphs and diagrams

Why python?

Python is a versatile and powerful programming language. It's open source and free for everybody, and has a huge community who constantly adds new features in the form of *libraries*. This means that python offers a vast set of tools that is constantly being updated and kept bug-free. But most importantly, despite being a full-fledged programming language, it's really easy to use. The syntax is quick and efficient, so you need few lines of code to achieve complex tasks. Finally, one of the big advantages of python are **Jupyter notebooks**. Jupyter is the programming environment that will be used throughout this book, and it's a real powerhouse: with it you can write code, display tables and images, alternate between blocks of formatted text and code, and much more. However it's only when one starts to use the language that the true extent of the applications becomes apparent. Upon reading this book you will find out where you can apply what you have learned, and start creating your own applications.

Prerequisites needed to follow this book

What you will *absolutely* need to follow through the examples given in the next chapters is:

- To know basic Microsoft Windows usage
- To know high school level calculus and linear algebra
- To know basic Excel usage

In addition, it would certainly help if you knew the following engineering-related modules:

- Solid mechanics;
- Theory of structures;
- Structural analysis and design;

These are not strictly necessary, as I will try to explain the technical stuff as clearly as possible, however the examples in this book rely heavily on these subjects.

Python basics

Setting up the environment

The first part of this book is intended to be a Python crash course: first we install all the software needed, then we move on to programming basics. The goal is to you with all the tools necessary to progress through the rest of the book, so it's very important that you follow through and understand everything that is written here. It will be useful, during the more technical parts subsequent to this one, to come back here every time you don't remember some particular Python syntax.

installing Anaconda

One of the easiest ways of getting python up and running on your computer is using the **Anaconda distribution**. Anaconda comes pre-packed with the most popular libraries for python (so you won't have to install them yourself), and also **Jupyter**, which is the notebook environment that we will use to write code. the instructions given here are intended for Microsoft Windows, but the same overall process applies for other operating systems. In order to install Anaconda:

- Open your web browser and navigate to https://www.anaconda.com/download and select your operating system;
- Select the version of Python that you wish to download. At the time this book was written, the latest version supported by Anaconda was Python 3.7. DO NOT download a version that uses Python 2, as some of the examples written in this book work only with Python 3;
- Once the installer has finished downloading, open it and follow the instruction to install Anaconda. I suggest leaving the default path as the destination folder.

Once Anaconda has finished installing, open Anaconda Navigator by typing "Anaconda navigator" in your computer's search bar. You will be greeted by a window that looks similar to the one in the next figure:
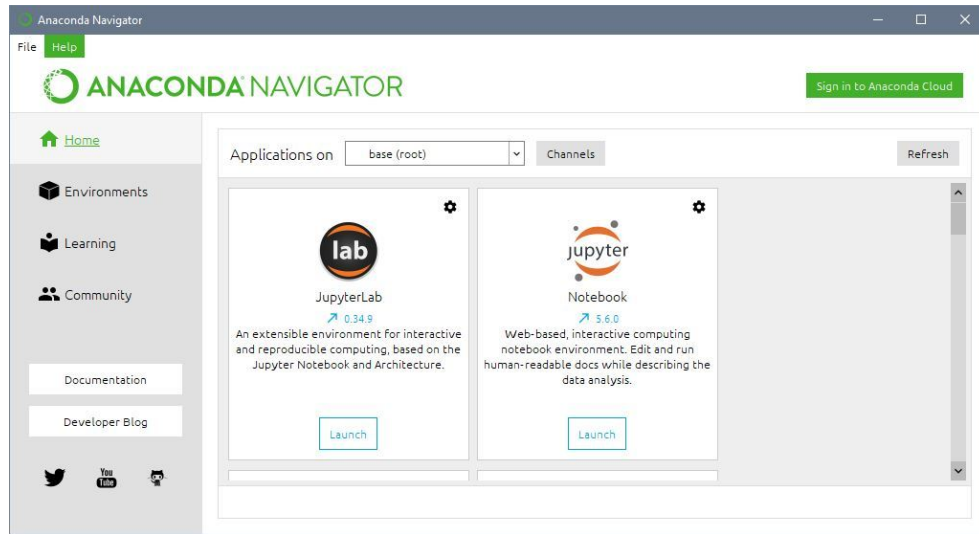
Anaconda navigator is essentially a hub from where you can access various Python coding environments. **Qt console** is a command line type of editor, **Spyder** is a scripting editor, and finally **Jupyter** is a notebook editor. This is the one we are going to use, so click *launch* to open up **Jupyter Notebook**.

This will open your default browser and display your computer's folder tree. In the address bar there should be written *http://localhost:8888/tree*. From this window you can navigate within the folders of your computer, open existing Jupyter notebooks, and create new ones.

Using Jupyter notebooks

To create a new Jupyter notebook, navigate to a folder of your choice and select*new>python3* from the top right-hand of the screen. This will create a new Jupter notebook in the folder you have selected. Your browser should then display something similar to what you see in the next image:
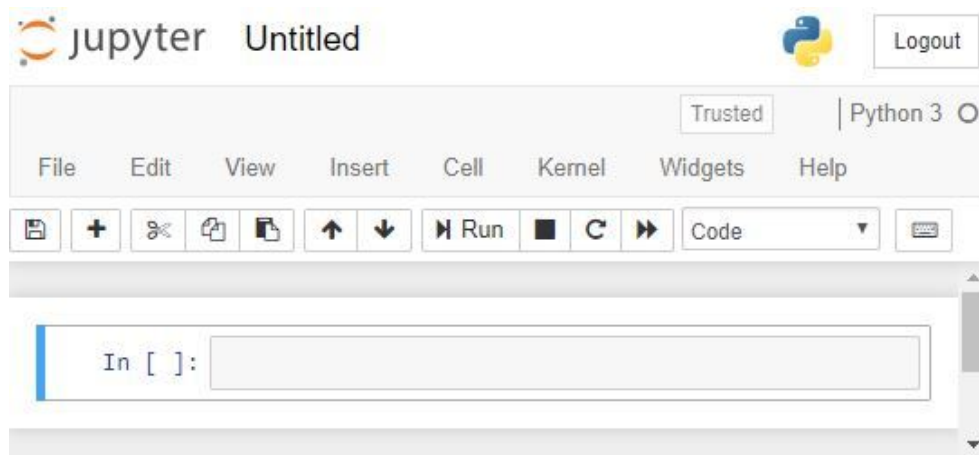
Jupyter is a **notebook environment**, which means that you have to write code inside *cells*. The frame in the previous image that says "In" is a cell. If the frame is highlighted in blue, it means that the cell is selected and ready to be edited. Before we start writing some code, let's go over the various tools and drop-down menus that Jupyter offers, so that you have a clear idea of how the interface works.

- from the *File* menu you can open, save, rename, export, and close notebooks;
- from the *Edit* menu you can copy, cut, merge, move, and delete cells;
- from the *View* menu you can customize the appearance of the interface;
- from the *Insert* menu you can insert a new cell above or below the one that is currently selected;
- from the *Cell* menu you can select various options to run the cells in the notebook. For example you can run a single cell, or the entire notebook. In addition, from this menu you can change the cell's type;
- from the *Kernel* menu you can manage the notebook's kernel. The kernel is basically the computational engine that runs the code written in the cells. It also stores all the variables you declare in your code;
- from the *Widget* menu you can manage the notebook's widgets, which are essentially extensions that can be installed to expand the capabilities of Jupyter;
- from the *Help* menu you can start a user interface tour (I suggest you try it), open various reference manuals and display the shortcut window.

The icons below the menu let you quickly select some of the tools present in the drop-down menus described above. Let's focus our attention on the little drop-down window that says "code": from here we can change the cell's type from **code** to **markdown**. Markdown is a typesetting language used to format your notebook. Usually, a well formatted notebook is made out of both *code* type cells and *markdown* type cells. The first contain all the various algorithms and calculations, the second contain text that explains what the code does, or that

displays the result of the \emph{code} cells. Let's leave the cell's type to \emph{code} for now, and click inside the frame of the cell. The frame turns green: we can now write some code inside it. Let's write

```
1  print("Hello World!")

 Hello World!
```

and click the *Run* button. Congratulation, you have written a python program! Notice that the **output** of the cell is displayed below the code. A cell does not necessary display an output: if there are no instructions to do so in the code, the cell will simply run without showing anything.

Now that we got the classic "hello world" example out of the way, we can start to explore the functionalities that python offers to its users. In the next few chapters you will learn how the syntax of the language works, and how to use some of the libraries that are available. It's **important**, if you have never used the libraries, that you follow through the examples given. This chapter will provide you with all the tools necessary to understand the code in the next more engineering-related parts of the book.

> **NOTE**
>
> From now on, when you wish to run one of the examples presented in this book, simply write the code in an empty cell and run it. I suggest writing the examples yourself, since simply copy-pasting them won't be much effective if you want to understand how they work.

Python syntax

Now we will go over the main elements that constitute the Python language.

Comments

Comments are lines of code that are ignored upon execution. They are used to comment and explain what the code does, in order to keep everything as clear as possible. The next code cell shows how to use them:

```
1   # This is a single line comment
2
3   '''This is a
4   multiline comment'''
```

as you can see python offers two ways of writing comments. The first one uses the \# symbol to write a single line comment, the second one uses the ''' sequence of characters to open and close a multiline comment.

Variables and datatypes

Variables are containers in which you can store values. The main types of data structures that are built-in in python (and that can be used without loading any library) are:

- **int**: integer numbers;
- **float**: floating point numbers;
- **bool**: boolean-type values (*True* or *False*);
- **string**: sequences of characters;
- **list**: python's built-in arrays;
- **dict**: Dictionaries;
- **tuple**: tuples;

Let's go ever them in detail.

Integer numbers

This datatype represents integers, meaning numbers without decimal values. Select an empty code cell and write the lines given in the next code snippet:

```
1 a=1
2 b=2
3 c=a+b
4 print(c)
```

```
3
```

We assign the values 1 and 2 to the variables **a** and **b**, then we sum them and store the result in variable **c**, and finally we print the value of **c**. **a** and **b** are integer values, and since **c** is the sum of two int-type objects, **c** it's an integer as well. As you may have guessed, the *print* function is used to display the value of whatever argument you put between the parentheses.

> **NOTE**
>
> Additions, subtractions and multiplications of integers will *always* result in an int-type result. Divisions between integers however will *always* result in a floating point value.

Floating point numbers

This datatype is used to represent, to put it mathematical terms, real numbers. They can be specified by using a decimal point, or using scientific notation. Let's take a look at the next piece of code:

```
1  a=1.0 #this is a floating point number
2  b=12.345 #this another floating point number
3  c=2.4e5 #this is floating point number
4          #specified using scientific notation
5  print(type(a))
6  print(c/b)
```

```
<class 'float'> 19441.069258809235
```

The function *type()* is used to retrieve the **dtype** of a certain object within the code. This function however does not print its output: we can achieve that by nesting the *type* and the *print* methods inside one another. As you can see, floating points numbers in python are called **floats**.

Boolean values

Boolean values are like switches, they can be either *True* or *False*. They are really useful when specifying **conditions** in your code, as will be explained later. The next code cell shows how to define boolean values:

```
1  a=True   #this is a bool value
2  b=False #this another bool value
3  print(type(a))

<class 'bool'>
```

Strings (str)

Strings are sequences of characters, and are used to represent **text**. You can open and close a string by using either the " characters or the "" characters. If you want to use one of these characters *inside* the string, simply add a backslash \ before the character you want to insert in the string. It's much better to understand these concepts by seeing them in action, so let's take a look at the next piece of code:

```
1  string_1="Hello"
2  string_2='World'
3  string_3="The quick 'brown' fox jumps \"over\" the lazy dog"
4  print(string_1)
5  print(string_2)
6  print(string_3)

 Hello
 World
 The quick 'brown' fox jumps "over" the lazy dog
```

As you can see in line 3, it's possible to use a ' character inside a string that has been specified with the " character. However if the string has been specified using " and we want to insert that same character inside it, we have to precede " with a backslash.

Now let's talk about string concatenation and formatting.

```
1  a="Hello"+" World"
2  b=2+2
3  print(a)
4  print("The result of 2+2 is: {0}".format(b))
```

```
Hello World
The result of 2+2 is: 4
```

Python lets the user **concatenate** strings using the plus sign, an operation that we will be using a lot throughout this book. In line 4 of code the previous code cell we see another very powerful feature: string formatting. With the *.format()* method python lets you insert variables within a string, in the places specified by the numbers between the curly brackets. The method *.format()* can accept multiple arguments, as shown in the next code cell:

```
1  a=12
2  b=3
3  print("a={0} and b={1}".format(a,b))
```

```
a=12 and b=3
```

This time we pass two variables to the format function that are inserted in the string at the positions specified by the curly brackets. Since **a** is the first argument that we pass to *.format()*, it will be inserted in the position specified by the curly brackets that contain 0, and since b is the second argument,it will be inserted in the position specified by the curly brackets that contain a 1.

String formatting can also be achieved with a different method that uses the % symbol:

```
1  a=2.3847567838
2  b=5
3  print("a=%.2f and b=%d"%(a,b))
```

```
a=2.38 and b=5
```

This time the positions in the string where **a** and **b** need to be inserted are identified by `%.2f` and `%d`. The letter `f` specifies a floating point number, and the letter `d` specifies an integer number. By writing `%.2f` we are telling python that

we want to display a floating point number stopping at the second decimal point.

This type of operation will be used a lot in the next chapters, as a way to better visualize the results of the calculations.

Lists (list)

Lists are used to represent a sequence of objects. If you have used other programming languages before, you can think of lists as arrays in which you can store any type of data you want. This includes integers, floating point numbers, strings, other lists, and so on. If you haven't used another programming language before, think of lists as **vectors** or **matrices**, depending on the dimension of the list itself. Let's now go over some examples, in order to understand how to create lists and how to use them.

In the next code cell we see some of the basic operations that involve lists:

```
1  myList1=[1,2,3,4] # this creates a list of integers
2  myList2=["hello", 2, 4.15, myList1]
3  mat=[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
4
5  print(myList1)
6  print(myList2)
7  print(mat)
```

```
[1, 2, 3, 4]
['hello', 2, 4.15, [1, 2, 3, 4]]
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

As you can see, lists are just collections of objects specified using square brackets and commas. You can nest lists one inside another to create multidimensional data structures, like you see in line 3 of the previos code snippet. Next, we explore how to retrieve data from lists:

```
1  a=[2, 4, 16, 32]
2  print(a[0]) #prints the first element of a
3  a[1]="hello" # assign 'hello' to the second position of a
4  print(a)
5  b=[[3,6],[9,12]]
6  print(b[0][1]) #prints element in position 0,1 of b
```

```
 2
 [2, 'hello', 16, 32]
 6
```

To access a value within a list, use the square brackets to specify the **index** of the element you wish to retrieve. In python, lists are indexed starting from zero, so the first element is in position 0, the second is in position 1, and so on. If you have a previously defined list and you wish to change one of its values, simply assign the new value to the position you want like you see in line 3 of the previous code cell. If a list is *multidimensional*, like the one defined in line 5, to access a specific position you need use two sets of square brackets (line 6).

Lists have also built-in methods that you can use to perform operations on their content. Let's explore the most useful of these functions:

```
 1  a=["dog", "cat", "lion"]
 2
 3  a.append("seal")
 4  print("append: {0}".format(a))
 5
 6  a.insert(2, "ant")
 7  print("insert: {0}".format(a))
 8
 9  a.remove("dog")
10  print("remove: {0}".format(a))
11
12  a.pop(3)
13  print("pop: {0}".format(a))
14
15  a.extend(["apple", "pear"])
16  print("extend: {0}".format(a))
```
```
 append: ['dog', 'cat', 'lion', 'seal']
 insert: ['dog', 'cat', 'ant', 'lion', 'seal']
 remove: ['cat', 'ant', 'lion', 'seal']
 pop: ['cat', 'ant', 'lion']
 extend: ['cat', 'ant', 'lion', 'apple', 'pear']
```

To access an object's built-in functions in Python we use "." followed by the name of the method. A "method" is just a built-in function of whatever object you are using (in this case a list). Confronting the input and output of the previous code panel you can get a pretty solid idea of what each of these

functions does, and I invite you to experiment with them in order to fully understand their behaviour. Let's go over them anyway, together with some other functions provided by python for lists:

- **append(*element*)**: Adds *element* at the end of the list;
- **clear()**: Deletes all the elements from the list;
- **copy()**: Returns a copy of the list;
- **count(*value*)**: Returns the number of elements equal to *value*;
- **extend(*iterable*)**: Attaches the iterable object specified at the end of the list. An iterable, in Python terms, is any object that can be seen as a collection of other objects. This includes, for example, lists and strings (a string is just a collection of characters);
- **index(*value*)**: Returns the position of the first element of the string that corresponds to *value*;
- **insert(*pos, element*)**: Adds *element* in the position specified by *pos*;
- **pop(*pos*)**: removes the element in the position specified by *pos*. If *pos* is not specified, removes the last element of the list;
- **remove(*value*)**: Removes the first element of the list that corresponds to *value*;
- **reverse()**: Reverse the order of the elements in the list;
  sort(): Sorts the elements in the list.

Dictionaries (dict)

Dictionaries are similar to lists, except that they are *unordered*. Each element in a dictionary has a **key** associated to it that acts like its name. To retrieve an element from a dictionary, you call the key associated with that element. Let's take a look at the next code cell:

```
1  mydict={"object" : "beam",
2          "material" : "steel",
3          "type" : "IPE120",
4          "h" : 120,
5          "b" : 64}
6
7  print(mydict["material"])
```

```
steel
```

Dictionaries are specified using curly brackets, and each **key** is associated with the corresponding **value**. When you want to access one of the values of the dictionary, you need to use its key. Since they have similar built-in methods as lists, we won't go over them again.

Tuples

Tuples are collection of objects that are **ordered** and **unchangeable**. This means that after you create a tuple you won't be able to modify its values in any way. the next code snippet shows how to create a tuple and use the only two methods provided for them by python:

```
1  mytuple=(1,2,3,"apple",5)
2  print(mytuple)
3
4  print(mytuple.count("apple"))
5  print(mytuple.index(5))
6
7  print(mydict["material"])

 (1, 2, 3, 'apple', 5)
 1
 4
```

As you can see, tuples are specified like lists, except they use round brackets instead of curly brackets. The two methods that can be used on a tuple are **count** and **index**. The first counts the number of elements within the tuple that correspond to the argument passed, the second returns the position of the first occurrence.

You might ask yourself what is the point of tuples if python already has lists. Well, tuples are used mostly when returning values from functions, and can be *unpacked* into separate variables. All of this will be explained later when we start talking about **functions**.

Conditions and statements

Python, like all other programming languages, gives the user the possibility of running pieces of code only when a specific condition is met. This is done

through **IF statements**, which evaluate a proposition and execute the next chunk of code based on whether that proposition was *true* or *false*. The next code cell gives an example of an IF statement:

```
1   x=3
2   y=5
3
4   if x<y:
5       print("x is smaller than y")
6   else:
7       print("x is greater than y")

 x is smaller than y
```

First, we assign values to variables **x** and **y**. Then the IF statement is introduced by using the keyword **if** followed by the condition *x<y*. If **x** is indeed smaller than **y**, then the first block of code is executed. If not, the block of code that follows the keyword **else** is executed. If there is no **else** keyword after the IF statement, and the condition is *False*, then the code written inside the statement will be ignored. This example also introduces a very important concept, explained in the next paragraph.

Python indentation

In other programming languages, the indentation of the code is merely a praxis to make it more readable. In python, indentation is instead used to specify blocks of code, and is *mandatory* for the code to work. Note that line 5 of the previous code snippet is indented: that's because it represents the block of code that the IF statement will execute if the condition **x < y** is *true*. Notice that a block of code will always be introduced by a semicolon, like we see at the end of line 4.

Python conditions

We find in python all the usual logical conditions found in math:

- Equals: **a == b**
- Not equals: **a != b**
- Greater than: **a > b**

- Greater than or equal to: **a >= b**
- Less than: **a < b**
- Less than or equal to: **a <= b**

Where a and b are two predefined variables.

> **NOTE**
>
> A common mistake that inexperienced programmers make is that they forget to use **two** equal signs to specify an *equals* condition. The single equal sign is used to assign values to variables, NOT as a conditional statement!

The ELIF statement and logical operators

What if we want to specify more IF statements one after another? This is what the ELIF statement is for:

```python
1  beam = {"mat" : "steel", "E" : 210000}
2
3  if beam["mat"] == "concrete":
4      print("This is a concrete beam")
5  elif beam["mat"] == "steel":
6      print("This is a steel beam")
7  elif beam["mat"] == "wood":
8      print("This is a wooden beam")
9  else:
10     print("I do not recognize the material")
```

```
This is a steel beam
```

The ELIF statements uses the same syntax as the IF statement, and can be called as many times necessary. The ELIF chain can be closed by an optional ELSE statement, that will be executed if all the conditions written before resulted to be *False*.

Next we learn about **logical operators**, used to combine multiple conditions together. Let's take a look at the next code cell:

```
1   a, b, c = 1, 2, 3
2
3   if a < b and b < c:
4       print("a<b<c")
```

```
a<b<c
```

Before discussing the IF statement, let's examine the first line: this is a shorthand notation used to assign a set of values to a set of variables. It simply stores 1 in **a**, 2 in **b** and 3 in **c**. Moving on, in line 3 we see the logical operator **and**. This operator returns *True* only when **both** the conditions associated with it are *True*. In this case, since the conditions *a<b* and *b<c* are indeed both true, the code inside the IF statement is executed. In python there are three logical operators:

- **and**: returns *True* if both statements are *True*;
- **or**: returns *True* if at least one of the two statements is *True*;
- **not**: used to reverse the result of a condition (e.g. **not(*True*)** will return *False*)

Shorthand notation

The IF statement can also be written using only one line of code. It's important that you know how to use this type of notation as well, although it's not strictly necessary. A lot of on-line examples use shorthand notation, so knowing how it works will certainly help you better understand them. Let's take a look at the next code cell:

```
1   x, y = 10, 5
2   if x>y: print("x>y")
3   print("x>y") if x>y else print("x<y")
4   print("x>y") if x>y else print("x=y") if x==y else print("x<y")
```

```
x>y
x>y
x>y
```

First, we assign two values to x and y, using the same shorthand notation that we already saw in code previously. In line 3 we see the shorthand notation for a single IF statement: it's the same as a normal IF statement, except we write the code in one single line. In line 3 we see the shorthand notation for the IF-ELSE statement. This one is a little bit different, because the order in which the condition and the code to execute are written is **reversed**. The first print command is executed if **x>y**, otherwise the code that follows the keyword *else* is executed. Having understood this, it should be easy to read the fourth line: this is the shorthand notation for a statement that normally would use the *elif* keyword. Instead of stopping the chain after the *else* keyword, we pass another shorthand IF statement that uses exactly the same notation as the one in line 3.

Python operators

An operator is simply a keyword that performs a certain operation on a given value. In Python there are seven types of operators:

- Arithmetic operators;
- Assignment operators;
- Logical operators
- Comparison operators;
- Identity operators;
- Membership operators;
- Bitwise operators

Arithmetic operators

Arithmetic operators are used to perform the basic mathematical operations that we all know. The next table gives a synopsis of their usage, for **x=13** and **y=4**

Operator	Name	Expression	Result
+	Addition	x + y	17
-	Subtraction	x - y	9
*	Multiplication	x * y	52

/	Division	x / y	3.25
%	Modulus	x % y	1
**	Exponentiation	x ** y	28561
//	Floor	x // y	3

> **NOTE**
>
> Notice that exponentiation is performed using two multiplication symbols, NOT using the ^ symbol like in other programming languages.

Assignment operators

Python assignment operators are a form of shorthand notation that can be used when we want to perform an operation on a single variable, and assign the result to *that same variable*. This type of operation is shown in code the next code cell:

```
1  x=3
2  print(x)
3  x=x+2 #the result of x+2 is stored in x
4  print(x)
```

```
3
5
```

Let's take a look at line 3. Python first performs the operation **x+2**, where **x** is equal to 3. Then the result of the addition is stored in **x**, that from this point on has a value of 5. Assignment operators perform this exact same task, except using less code. An example is given in the next code snippet:

```
1  x=3
2  print(x)
3  x+=2 #the result of x+2 is stored in x
4  print(x)
```

```
3
5
```

In line 3, += represents the addition assignment operator. Python provides the user with assignment operators for all the major mathematical operations. Given **x=3**, the most important are:

Operator	Expression	Equivalent	Result
+=	x += 2	x = x + 2	5
-=	x -= 2	x = x - 2	1
*=	x *= 2	x = x * 2	6
/=	x /= 2	x = x / 2	1.5
%=	x %= 2	x = x % 2	1
**=	x **= 2	x = x ** 2	9
//=	x //= 2	x = x // 2	1

Comparison and logical operators

We have already seen these kinds of operators when we talked about conditions. Please refer to that paragraph for an explanation of their usage.

Membership operators

Membership operators are used to check if a certain subset of values is present in a larger collection. For example, they can be used to see if a certain number appears in a given list. Let's see what the next code snippet does to get a clear understanding of this type of operator:

```
 1  a=3
 2  b=12
 3  myList=[1,2,3,4,5]
 4
 5  if a in myList:
 6      print("a is in myList")
 7  else:
 8      print("a is not in myList")
 9
10  if b in myList:
11      print("b is in myList")
12  else:
13      print("b is not in myList")
```

```
a is in myList
b is not in myList
```

As you can see, we use the keyword **in** to check if **a** and **b** appear in **myList**. In this case both **a** and **b** are integer numbers, however membership operators work also with strings, dictionaries and tuples. If python finds an occurrence of the specified value inside the collection, it will return *True*. If you want to check if a particular subset of values *does not* occur inside a collection, you can use the keyword **not in** instead, that will return *False* when said subset is found.

For loops

For loops are used to iterate through a sequence, be it a list, a string, a dictionary, etc., or to perform a repetitive operation a certain number of times. A *for* loop is introduced with the keyword **for**, followed by an *iterator*, followed by an iterable object. The block of code that comes after that is run again and again at each iteration of the *for* cycle. Let's take a look at the enxt code cell:

```
 1  myList=[2,4,8,16]
 2
 3  for i in myList:
 4      print(i)
```

```
2
4
8
16
```

First, we create an iterable object, in this case a list called *myList*. Then we iterate through that object using a *for* loop: in this example, at each iteration the variable **i** (the *iterator*) changes its value to match the one in **myList**. Since **myList** has a length of 4, the *print* command is executed four times, one for each iteration of the *for* cycle.

The *range()* function

The *range* function is used to create a sequence of numbers that go from 0 to a certain value, and can be very useful in *for* loops as a way to repeat an operation a specific number of times. But actions speak louder than words, so let's take a look at the next code cell:

```
1  for i in range(3):
2      print(i))
```

```
0
1
2
```

If we pass the *range* function the value 3, the variable **i** takes a value that goes from 0 to 2, in other words 3-1.

> **NOTE**
>
> The *range(n)* function creates a sequence that goes from 0 to n-1, NOT from 0 to n or from 1 to n.

You can use range to cycle through the members of a list:

```
1. myList=[2,4,8,16]
2. list_length=len(myList)
3. for i in range(list_length):
4.     print(myList[i])
```

```
2
4
8
16
```

Let's break this down line by line. First, we create a list called *myList*. Then in line 2 we use a function that we haven't met before, the **len** function. What **len** does is return the *length* of any iterable object that you pass to it. In this case, since **myList** has four elements in it, **len** will return the value 4, which is then stored inside the variable **list_length**. When we invoke the *for* loop in line 3, we pass **list_length** to the *range* function, thus creating a *for* loop that iterates from 0 to 3. The iterator **i** can then be used to access each element of **myList** one after the other.

This is very common modus operandi for any programming language, so please take the time to fully understand the previous code snippet.

The *break* and *continue* statements

Using the **break** statement we can stop a loop at a certain iteration:

```
1   for i in range(5):
2       if i==3:
3           break
4       print(i)

0
1
2
```

As you can see, the loop gets interrupted as soon as **i** is equal to 3. Any code inside the *for* loop that comes after the **break** statement is ignored.

The **continue** keyword is instead used to jump to the next iteration of the cycle:

```
1   for i in range(5):
2       if i==3:
3           continue

0
1
2
4
```

This time when **i** is equal to 3 the print order for that iteration is ignored (because it comes after the *continue* statement) and the program jumps at the next iteration, where **i=4**.

Functions

The last element that we are going to discuss in this brief introduction to Python is **functions**. Even if you haven't programmed before, you may be familiar with the concept of function thanks to other subjects like math. A function is an element that takes some inputs, performs operations on them, and returns an output. When writing an algorithm, functions are useful because they save a lot of lines of code, and help to *conceptualize* what the various elements of the program do. For example, say you are writing a program that calculates the deflection of a beam. Well, you need to know the moment of inertia of the beam's section in order to do that. The best way to approach this problem is to write a function that takes the beam's geometrical properties as inputs, uses them to calculate the moment of inertia, and outputs the result. So every time you want to calculate a moment of inertia you don't need to write the whole formula, you can simply call the function that you have previously written.

Now let's see a few examples written in python. In the next code cell we write a function that calculates the area of a trapezoid:

```python
1  def area(B, b, h):
2      return (B+b)*h/2
3
4  myArea = area(3,1,2)
```

```
4.0
```

A function is defined using the keyword **def**, followed by its **name**, followed by the function's **inputs**. In this case, we create a function called **area** that takes three inputs:

- **B** is the bottom base of the trapezoid;
- **b** is the top base of the trapezoid;
- **h** is the height of the trapezoid.

Notice that after the inputs there is a semicolon. This introduces the indented block of code that is executed when the function is called, in this case the formula of the trapezoid's **area**. The keyword **return** is used to specify what's

the output of the function: whatever follows that keyword is what the function will **return**. In line 4 we see how a function is *called*. Simply write the name of the function specified previously followed by the arguments. The output of the function is stored in a variable called **myArea**, which is later printed using *print*.

> **NOTE**
>
> don't be confused by the fact that in code \ref{code:functions1} the variables **B**, **b** and **h** don't have a value assigned to them. The variables used within a function's definition only get assigned a value when the function is called later in the code, in this case at line 4.

Variable scope

The **scope** of a variable is the space within the code where that variable is accessible. It's very important to understand this concept, otherwise it will become the source of many errors later on. In python, and in many other programming languages, variables can be distinguished in two groups: **global** and **local**. Global variables are accessible from everywhere in the code, local variables are accessible **only** within the block of code where the have been declared. For example, if you declare a variable inside a function, the value stored in that variable will only be accessible from *within the function*. This means that if you try to run the following block of code you will get an error:

```
1  def myFunc(a):
2          myVariable = 1
3      return a**2
4  res=myFunc(4)
5  print(myVariable)
6  print(res)
```

This happens because the *print* function is trying to access the value of **myVariable**, but **myVariable** has been declared inside a function definition, and therefore cannot be accessed outside of it. Let's see what this other piece of code does:

```
1  myVariable = 5
2  def myFunc(a):
3          myVariable = 1
4      return a**2
5  res=myFunc(4)
6  print(myVariable)
7  print(res)

 5
 16
```

This might seem strange: first we store 5 inside **myVariable**, then we change that value to 1 in line 4. So how come when we call *print(myVariable)* python displays 5? Because the **myVariable** declared outside of **myFunc** is a completely different object from the **myVariable** declared inside of it. They have completely different memory allocations, and after the block of code written inside **myFunc** has finished running, the **myVariable** declared inside of it is destroyed. In this example, the **myVariable** created in line 1 is a *global* variable, and the **myVariable** created in line 3 is a *local* variable.

Function parameters

The **parameters** of a function are what you put between the parentheses when you first declare it. When you later call the function from somewhere else in the code, you need to specify a value for *all* the parameters that appear in the function's definition, otherwise you will get an error. But what if you want a certain parameter to have always a specific value that can be changed to something else only when needed? This is what **default value parameters** are for. the next code cell shows how to use them:

```
1  def myFunc(radius, perimeter=False):
2      if perimeter==False:
3          return 3.14*radius**2
4      else:
5          return (3.14*radius**2, 2*3.14*radius)
6
7  print(myFunc(4))
8  print(myFunc(4, perimeter=True))

 50.24
 (50.24, 25.12)
```

We want to create a function that returns the area of a circle, and when needed, its perimeter as well. In line 1 we define a function called **myFunc** that takes the obligatory parameter **radius** and the optional parameter **perimeter**, by default set to *False*. As you can see it's really easy to specify default parameters, all you have to do is give them a default value in the function definition. Inside the body of the function there is an IF statement that is used to check whether the user has set **perimeter** to *True* while calling the function. If not, and **perimeter** is set to *false*, we simply return the area of the circle using the formula $A=3.14r^2$. If instead **perimeter** is set to *True*, in line 5 we return a tuple that contains the area in the first position, and the perimeter in the second. Tuples are useful when you want a function to return more than one single value. When we call **myFunc** in line 7 specifying only the **radious** parameter, the function returns only the area of the circle. But if we call **myFunc** setting **perimeter** to true, the function returns the tuple specified in line 5.

Unpacking

When a function returns a tuple, you can **unpack** its contents and store each entry of the tuple into separate variables. Using The same function called **myFunc** defined previously, the output can be unpacked like this:

```
1  area, perimeter=myFunc(4, perimeter=True)
2  print(area)
3  print(perimeter)

 50.24
 25.12
```

In line 1, when calling **myfunc**, the output is automatically unpacked inside the variables **area** and **perimeter**.

Conclusions

You now have all the knowledge you need to read and understand most of the syntax that you will ever find in a python code. However, the vast majority of python programs use **libraries** to expand the capabilities of the language, and indeed libraries are what you will be using in every example of this book. In the next chapter you will learn how to use probably the most widely-known library of python, **numpy**.

Numpy

In this chapter you will learn how to use a library called **NumPy**, which is the most widely used tool to manage operations that involve *arrays*. Virtually *every* python program uses this library in some way or another. Since it is, as you might say, the "industry standard" in every branch of science and engineering that uses python, you can rest assured that it will stay relevant for many years to come. In this chapter we will first go through a quick introduction to NumPy that pinpoints the functionalities of the library that are most useful, then a practical example is provided.

The NumPy library

As previously stated, the NumPy library is mainly used to create and edit **arrays**. An array is a data structure similar to a **list**, with the difference that it can contain only one type of object. For example you can have an array of integers, an array of floats, an array of strings and so on, however you can't have an array that contains *two* datatypes at the same time. But then why use arrays instead of lists? Because they are faster, they have more functionalities, and they better represent mathematical concepts like vectors and matrices.

What NumPy does is provide the user with a powerful object called **ndarray** (but generally referred to simply as *array*) that expands the functionalities of Python quite a bit.

Importing the library

In Python, when you want to use an external library like NumPy, you have to **import** it before you can access its functionalities. Usually you would also need to install it first, but luckily Jupyter has all the libraries that we need pre-installed. Open Jupyter, create a new jupyter notebook, write the following code in an empty cell and run it:

```
1  import numpy as np
2  a=np.zeros((4))
3  print(a)
```
```
[0. 0. 0. 0.]
```

To load a library, we need to use the keyword **import**. Then we can specify an abbreviation for that library that will be used throughout the code, using the keyword **as**. In line 1 we import *numpy*, and we assign the nickname "np" to it. This way it will be enough to write "np" followed by whatever function you need in order to acces the library's functionalities. In this example we create an array of zeros using NumPy's built in method *zeros*, that takes as input a tuple that specifies the dimensions of the array to be created. We want a one-dimensional array of four elements, so the tuple will just contain 4.

Creating arrays

You can create NumPy arrays from lists and tuples with the function **array()**. Let's see how it works:

```
1  a=np.array([1,2,3,4])
2  b=np.array([[1,2],[3.5, 4.2]])
3  print("a={0}".format(a))
4  print("b={0}".format(b))

 a=[1 2 3 4{]}
 b=[[1. 2. ]
 [3.5 4.2]]
```

In line 1 we pass the *array* function a lists of integers, so NumPy creates an array of integers. In line 2 we pass a multidimensional list that contains both integers and floats, and as you can see NumPy automatically converts the *int* values to *float*, because an array can contain only one type of data}.

If you know the dimensions of an array that you are going to need, but you want to populate its positions later, you can create arrays of **zeros** or **ones**:

```
1  a=np.zeros((3))
2  b=np.ones((2,3), dtype=int)
3  print("a={0}".format(a))
4  print("b={0}".format(b))

 a=[0. 0. 0.]
 b=[[1 1 1]
 [1 1 1]]
```

The functions `ones()` and `zeros()` take as inputs a tuple that specifies the dimensions of the axes of the array. In line 2 we pass the tuple (2,3), in order to create an array of ones with two lines and three columns. Both `zeros()` and

`ones()` can take the additional parameter *dtype,* which is used to specify the dtype of the elements in the array. When we create **b** we specify *int* as the dtype, so **b** will be composed of integer values.

To create an array that contains a sequence of numbers that goes from a starting value to an ending value, you can use the `linspace()` function:

```
1  a=np.linspace(1,10,6)
2  print("a={0}".format(a))

 a=[ 1.  2.8 4.6 6.4 8.2 10. ]
```

linspace takes as arguments the starting point, the ending point and the number of elements of the sequence that you want to create. In the previous code cell we create an array that goes from 1 to 10, composed by 6 evenly spaced values.

A function similar to `linspace()` is `arange()`. Instead of specifying the number of elements between the start and end positions, with `arange()` you need to specify a **step**:

```
1  a=np.arange(1,10,1)
2  print("a={0}".format(a))

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

Retrieving an array's dimension and dtype

When you need to retrieve information about an existing array, NumPy offers the following methods:

- **ndarray.ndim:** returns the number of axes of the array;
- **ndarray.shape:** returns the dimensions of the array as a tuple. Essentially it returns the number of elements in each axis;
- **ndarray.size:** the total number of elements in the array;
- **ndarray.dtype:** returns the dtype of the elements of the array.

Let's see these methods in action:

```python
1  a=np.array([[1,2],[3,4]])
2  print(a.ndim)
3  print(a.shape)
4  print(a.size)
5  print(a.dtype)
```

```
2
(2, 2)
4
int32
```

As you can see, **a** is composed by *int32-type* objects. This is one of the various *int-type* objects provided by NumPy, and it represents a 32-bit long integer. You generally don't need to worry about which type of integer or float NumPy is using, as long as you recognize that it is indeed an *int* or *float*.

How to access and modify the elements of an array

NumPy arrays are indexed starting from zero like **lists**, however the syntax used to access a certain value within an array is a little bit different. Let's take a look at the next code cell:

```python
1  a=np.array([2.45, 3.75])
2  a[0]=16
3  b=np.array([[0,1],[2,3]])
4  print(a[0])
5  print(b[1,1])
```

```
16.0
3
```

To access an element of an array we use the square brackets like we did for **lists**. Notice that when we need to retrieve a value from a two-dimensional array like **b**, we specify the position of the element using *only one set of square brackets*, in this manner:
[i,j]
If you remember, for lists the same operation was made using this syntax instead:
[i][j]

Be careful not to confuse the two! As we did for lists, we can change the value stored at a certain position of the array, like we see in line 2.

Mathematical operations between arrays

NumPy performs mathematical operations between arrays in a *element-wise* fashion. This means that the arrays must all have the same number of elements in them, otherwise you will get an error. The next code cell shows some of the possible operations that can be performed:

```
1  a=np.array([1,2,3,4])
2  b=np.array([2,4,8,16])
3  print(a+b)
4  print(a-b)
5  print(a*b)
6  print(a/b)
7  print(a**2)
8  print(np.sqrt(b))
```

```
[ 3 6 11 20]
[ -1 -2 -5 -12]
[ 2 8 24 64]
[0.5 0.5 0.375 0.25 ]
[ 1 4 9 16]
[1.41421356 2. 2.82842712 4. ]
```

If you have used Excel before, you may have noticed that the operation in line 3 is smilar to summing two columns of a spreadsheet. These kinds of operations between arrays are very common, and indeed very useful. but what if you wanted to perform the **matrix** product instead of the element-wise multiplication? Then you would use @ instead of * as the multiplication symbol:

```
1  a=np.array([[1,2],[3,4]])
2  b=np.array([[2,4],[8,16]])
3  print(a@b)
```

```
[[18 36]
 [38 76]]
```

Array slicing

Array slicing is used to perform operations only on certain parts of a given array. The next code cell shows the most commonly used slicing operations for one-dimensional arrays:

```
1  a=np.linspace(0,10,11)
2  print(a)
3  print(a[2:5])
4  print(a[1:8:2])
```

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

As you can see we can use semicolons to print only the elements between a starting point and an ending point an an array. Moreover, in line 3 we see how to select elements at certain intervals: the syntax *1:8:2* means "take every second element of **a** between the positions 1 and 8".

For multidimensional arrays the syntax is the same, except each axis of the array can be sliced independently. To understand what this means, let's go over the next code cell:

```
1  mat=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
2  print(mat[:,2])
3  print(mat[3,:])
4  print(mat[:,2:4])
```

```
[ 3 7 11 15]
[13 14 15 16]
[[ 3 4]
 [ 7 8]
 [11 12]
 [15 16]]
```

In line 2 we select all the lines using the semicolon, and only the third column by passing 2 as the positional index for the second axis. Line 3 is similar, except it selects the fourth line of the matrix. The print order in line 3 displays the third and fourth columns instead.

NOTE

You may have noticed that when slicing using the semicolon, NumPy cuts the array from the starting position to the ending position *excluded*. This is why line 4 of the previous code cell does not give an error, even tough the index 4 is out of bounds for an array with 4 columns. (remember that arrays are indexed from 0!)

Array union and concatenation

The functions *hstack* and *vstack* can be used to merge two or more arrays:

```
1  a=np.array([1,1])
2  b=np.array([2,2])
3  c=np.array([3,3])
4  print(np.hstack((a,b,c)))
5  print(np.vstack((a,b,c)))
```

```
[1 1 2 2 3 3]
[[1 1]
 [2 2]
 [3 3]]
```

and *append* and *insert* can be used to add new elements inside the array:

```
1  a=np.array([1,2,3])
2  print(a)
3  a=np.append(a,4)
4  print(a)
5  a=np.insert(a,2,7)
6  print(a)
```

```
[1 2 3]
[1 2 3 4]
[1 2 7 3 4]
```

As you can see, *append* adds an element at the end of an array, and *insert* adds an element at a certain position. The second argument of *insert* can be an iterable object (like a list or another array) that specifies more than one position where to add the element specified by the third argument. Both *append* and *insert* take the optional argument **axis**, used to specify the axis along which the operation must be performed.

NumPy offers many other functionalities on top of the ones explained in the previous sections. So many, in fact, that they would require a separate book to be explained in their entirety. In the next chapters we will introduce some of them, however we will focus our attention only on the ones that are most useful and relevant to the practical goals that we have.

EXAMPLE: calculating the internal forces of a beam using NumPy

We can finally move on to the first practical example of this book. In this section many of the concepts that have been explained before will come together, hopefully giving you a hint of their possible practical applications. Since this is the first time we use Python (and Jupyter) in a more structured way, the following example will be very simple from an engineering point of view, and yet very useful to understand the functionalities explained in the previous chapters.

Problem definition

The goal of this example is to calculate the bending moment and shear of simply supported beam using NumPy. The beam is shown in the next figure:
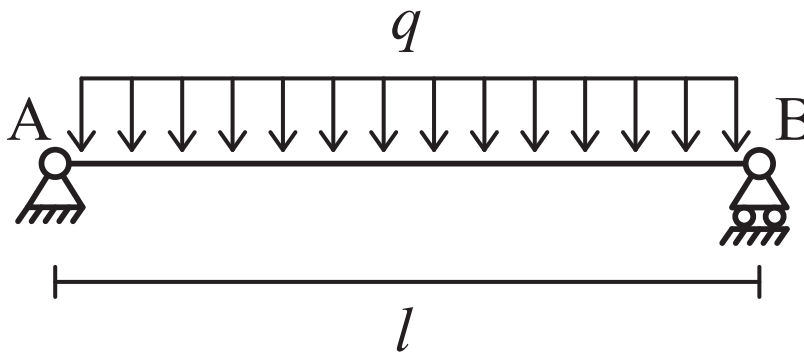
Where:

- q=20 kN/m
- l=5 m

How to input the problem's quantities in Python

The first step to solve many problems using a computer is simply to store the various given quantities in **variables**, so that they can be used throughout the

code. But before we can do that, we have to load NumPy, so we can use its functionalities:

```
1   import numpy as np
```

It's common practice to put all your library-loading code lines in the first cell of the notebook, to keep it separated form the rest. Remember to run this cell at least once *before* you run other cells that contain NumPy functionalities, otherwise you will get an error!

In the next cell, we store the length of the beam and the value of the distributed load in two variables:

```
1   l=5  #m
2   q=20  #kN/m
```

Notice how each quantity has its own unit of measure written as a comment next to it. This may seem a bit superfluous given the simplicity of the problem, but it will most certainly become a necessary habit when tackling more difficult applications. It's common practice to keep code inputs in their own cell as well, so that you know where to find them in case you need to modify them. From this point on, however, it's up to you to decide how to organize your code. Keep in mind that the easier it is to understand your code, the easier it is to find bugs.

Calculating the bending moment and shear

The beam's ground reactions and coordinate system are shown in the next figure:
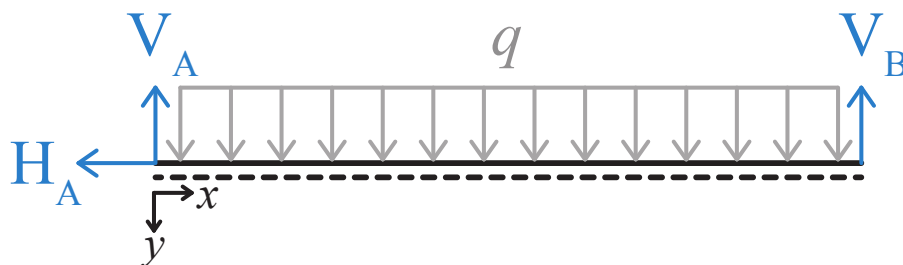
The equilibrium equations are:

$$\begin{cases} H_A = 0 \\ V_A + V_B - ql = 0 \\ V_B l - \dfrac{ql^2}{2} = 0 \end{cases}$$

solving the system we obtain

$$H_A = 0 \qquad V_A = \frac{ql}{2} \qquad V_B = \frac{ql}{2}$$

so the bending moment **M** and the shear **V** of the beam will be equal to

$$M = V_A x - \frac{qx^2}{2} = \frac{q}{2}(lx - x^2)$$

$$V = V_A - qx = q\left(\frac{l}{2} - x\right)$$

Our goal is to translate all of these equations in Python language, and obtain two vectors named **M** and **V** that contain the value of the bending moment and the shear of the beam for **x** that goes from A to B. All of this is done in the next code cell:

```
1  x=np.linspace(0,l,20)
2
3  M=q/2*(l*x-x**2)
4  V=q*(l/2-x)
```

```
[ 0. 12.46537396 23.54570637 33.24099723 41.55124654
 48.47645429 54.0166205 58.17174515 60.94182825 62.32686981
 62.32686981 60.94182825 58.17174515 54.0166205 48.47645429
 41.55124654 33.24099723 23.54570637 12.46537396 0. ]
```

In line 1 we create the x coordinate using the NumPy function **linspace**. As you can see we pass l as the ending value and 20 as the number of steps, meaning that NumPy will create a vector of evenly spaced numbers that go from 0 to l, which in this case is equal to 5. From this point on it's just a matter of copying the formulas: NumPy will perform all the operations *element-wise*, so **M** and **V** will be arrays composed of 20 elements each, corresponding to every coordinate value contained in **x**.

If you are familiar with Excel, you can think of x as a column of values to which you apply the formulas in lines 3 and 4, in order to obtain two new columns containing the values of **M** and **V**.

Later in this book we will learn to use **Matplotlib** to plot various graphs and diagrams. For now, just copy and paste the following code in an empty cell to display **M** and **V**:

```python
from matplotlib import pyplot as plt
plt.figure(figsize=(10,4))
plt.plot([0]*len(x), color='k')
plt.plot(-M)
plt.plot(V)
```

The result will be a graph similar to this:
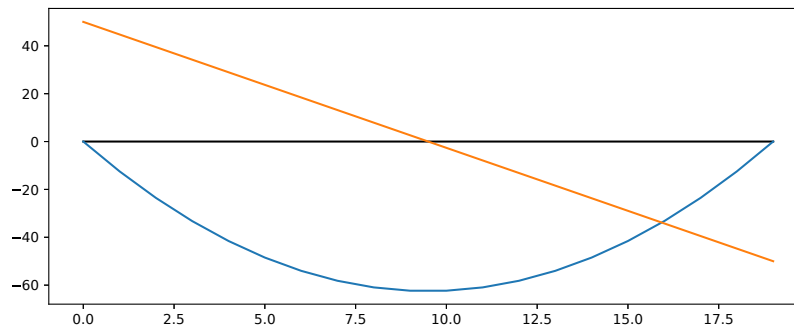
the graph is still pretty basic, but it is enough to understand that the values of **M** and **V** are correct.

Conclusions

Numpy is one of python's most useful libraries. Throughout this book you will encounter it often, and the same goes for pieces of code that you might find on the internet. This is especially true for engineering-related problems, where the need to manage vectors and matrices is very common. Next, you will learn how to perform symbolic calculations using **sympy**.

Sympy

Computer Algebra Systems

Computer Algebra Systems (*CAS*) allow the user to perform symbolic calculations. What we have seen so far were only *numerical calculations*, because when evaluating a formula each variable had a value assigned to it so that it could return a numerical result. Suppose we had a function like this one:

$$f(x) = 3x^3 + 4x^2 - 1$$

and we wanted the computer to calculate the derivative for us, keeping x as a symbol. Well, this is exactly what Computer Algebra Systems are for.

In Python the most widely used CAS is **SymPy**. It can store formulas and variables as symbolic expressions, and perform symbolic operations on them. Given f(x), SymPy would be able to calculate f'(x) as an expression, returning

$$f'(x) = 9x^2 + 8x$$

SymPy is a vast and powerful library, and in this book we will only go through its most useful and powerful functionalities. At the end of this chapter you will have at your disposal a solid set of tools to perform symbolic calculations.

Importing the library

As we did when we first imported NumPy in the previous chapter, we're going to import SymPy specifying an abbreviation for it. In an empty cell write the following code:

```
1  import sympy as sp
2  sp.init_printing(use_latex="mathjax")
3  display(2+sp.sqrt(3))
4  display(sp.sqrt(32))
```

$$\sqrt{3} + 2$$

$$4\sqrt{2}$$

In line 1 we import the library, and in line 2 we tell Jupyter how to print SymPy's expressions. More specifically, we want the expressions to be printed in LaTeX format, and we want to use *Mathjax* to render LaTeX. Lines 3 and 4 use the command **display** to print two SymPy expressions. The output will be written in a beautiful LaTeX format. We could use **print** instead of **display**, but then the output wouldn't be as pretty and easy to read.

> **NOTE**
>
> LaTeX is a typesetting language, and is in fact the lingua franca for creating beautiful expressions and scientific documents. Don't worry if you have never used it, for now it's just a way to display prettier outputs in the cells. The second part of this book will give more informations about LaTeX in Jupyter notebooks.

Symbols

SymPy lets the user define the symbols that will be used during symbolic calculations. You can think of them as undefined variables, created using the command `symbols()`.

```
1  a, b=sp.symbols("a b")
2  expr1=a+b
3  expr2=expr1**2
4  display(expr1)
5  display(expr2)
```

$$a + b$$
$$(a + b)^2$$

In line 1 we create two symbols, **a** and **b**, by passing the `symbols()` function a string that contains the names of the variables. Each symbol in the string is separated by a space and corresponds to one of the variables on the left of the equal sign. From this point on, the variables **a** and **b** will be *symbolic*, and any operation that involves them will return a symbolic result as well.

The *subs()* command and numerical evaluation

Suppose that you have created a symbolic expression, and you want to substitute its symbols with numbers. This kind of operation is done using the *subs()* method: let's see how to use it.

```
1  x, y=sp.symbols("x y")
2  f=x**2+y**2+sp.Rational(1,3)*x*y
3  display(f)
4  res1=f.subs(x,1)
5  display(res1)
6  res2=f.subs([(x,1), (y,5)])
7  display(res2)
8  display(res2.evalf(5))
```

$$x^2 + \frac{xy}{3} + y$$

$$y^2 + \frac{y}{3} + 1$$

$$\frac{83}{3}$$

$$27.667$$

After creating the symbols **x** and **y**, we define the expression **f** as seen in line 2. The function *sp.Rational* is used to specify a rational number, in this case 1/3. Now we want to substitute 1 to **x**: in order to do this we use the *.subs()* method, specifying the variable that we want to substitute and its value. This operation results in a new expression that is stored in **res1**. Note that when **res1** is printed out in line 5, **x** has been substituted with 1. In line 5 we do the same thing again, except this time we substitute both **x** and **y**. This is done by passing the *.subs()* command a list of tuples containing the variables and the values in pairs. Finally we use the *.evalf()* command to collapse the whole expression in one single *float* value. The argument of *.evalf()* specifies the number of digits of the output.

Calculus

Derivatives and integrals are very common operations in engineering, so let's see how SymPy deals with them:

```
1  x=sp.symbols("x")
2  expr=x**2+sp.Rational(1,2)*x-5
3  display(expr)
4  derivative=sp.diff(expr,x)
5  display(derivative)
6  integral1=sp.integrate(expr,x)
7  display(integral1)
8  integral2=sp.integrate(expr,(x, 0, 3))
9  display(integral2)
```

$$x^2 + \frac{x}{2} - 5$$

$$2x + \frac{1}{2}$$

$$\frac{x^3}{3} + \frac{x^2}{4} - 5x$$

$$-\frac{15}{4}$$

The code is pretty self-explanatory. To perform the derivative of **expr** with respect to **x**, simply pass **expr** and **x** as arguments to `sp.diff()`, as shown in line 4. The indefinite integral is performed using `sp.integrate()`, and the arguments are the same as `sp.diff()`. To calculate the definitive integral you need to group the integration variable with the two extremes of integration in a tuple. Line 8 show an example of this kind of operation, where we integrate **expr** with respect to **x** between 0 and 3.

Solvers

Solvers are one of the most powerful tools of SymPy.They are used to solve various types of equations, something that engineers often need to do in their professional work.

Equations

In SymPy, equations are defined using the `Eq()` command. These equations can then be used as inputs for the various solvers that SymPy provides in order to find the solutions. Note that it's **wrong** to specify equations using = or ==.

```
1  x=sp.symbols("x")
2  my_eq=sp.Eq(x**2-4,0)
3  display(my_eq)
4  sp.solveset(my_eq,x)
```

$x^2 - 4 = 0$
$-2, 2$

In the previous code snippet we create an equation called **my_eq**, and then we pass it to *solveset()*, which is the main solver of SymPy.

> **NOTE**
>
> It is often superfluous to use `Eq()` to define an equation. Most of the times it's faster to write the equation as $f(x)=0$ and pass $f(x)$ straight to the solver. By default, if the solver receives an **expression** as input, it will automatically add =0 at the end of it, and solve the equation accordingly.

The *solveset* command

The main function that SymPy uses to solve algebraic equations is **solveset**, to which the user can pass an equation written in the form of an **Eq** instance or an expression that will automatically assumed to be equal to 0. The next code cell gives two examples:

```
1  x=sp.symbols("x")
2  res1=sp.solveset(sp.cos(x)*2, x)
3  display(res1)
4
5  res2=sp.solveset(sp.Rational(2,5)*x+3, x)
6  display(res2)
7  display(res2.args[0])
```

$$\left\{2n\pi + \frac{\pi}{2}|n \in \mathbb{Z}\right\} \cup \left\{2n\pi + \frac{3\pi}{2}|n \in \mathbb{Z}\right\}$$

$$\left\{-\frac{15}{2}\right\}$$

$$-\frac{15}{2}$$

Line 2 shows how to use `solveset()` to solve the simple equation $2\cos(x)=0$. Notice how the cosine is specified using the SymPy function `sp.cos()`, and how we don't need to add =0 at the end of the expression. The lines 5 and 6 show how to solve an equation and retrieve the result using the `.args` argument. But why use `.args` instead of writing, say, `res2[0]`? Because the results of `solveset()` aren't lists, but rather data structures defined by the library itself. In the example above, the result is what SymPy calls a **FiniteSet**, which isn't in itself an iterable object. To acces the values stored in a FiniteSet, you need to use `.args`, and pass the index of the result you want to retrieve.

Systems of equations: *linsolve* and *nonlinsolve*

SymPy can of course also solve systems of equations. It gives the user the ability to solve linear systems using `linsolve()`, and nonlinear systems using `nonlinsolve()`. The syntax is very similar *solveset*, only you need to pass a list of equations instead of just one and a tuple containing the

unknown terms. The next code cell gives an example of how to use these two modules:

```
1  x, y=sp.symbols("x y")
2  res1=sp.linsolve([x+y-4, x-y-9], (x,y))
3  display(res1.args[0])
4
5  res2=sp.nonlinsolve([2*x**2+3*x-y-1, 3*x+2*y-5],(x,y))
6  display(res2)
```

$$\left(\frac{13}{2}, -\frac{5}{2}\right)$$

$$\left\{\left(-\frac{9}{8}+\frac{\sqrt{193}}{8}, -\frac{3\sqrt{193}}{16}+\frac{67}{16}\right), \left(-\frac{\sqrt{193}}{8}-\frac{9}{8}, \frac{3\sqrt{193}}{16}+\frac{67}{16}\right)\right\}$$

For the scope of this book, the tools presented in the section above are more than enough. Next up, we will learn how to use *pandas* to manage data structures.

Pandas

Up until now we have encountered four types of data structures: lists, dictionaries, tuples and numpy arrays. These are useful when the data they handle is relatively simple, especially numpy arrays are designed to work **fast** with regular matrices. But what if you need to import data from an Excel spreadsheet? or if you want to structure, filter and categorize the data contained in a table? Well, that is exactly what **pandas** is designed to do: handle big, complex data structures with ease.

The main tool that pandas provides to the user is a new type of structure called a **DataFrame**. Think of it like a 2D data structure with headers for both columns and rows. It allows the user to perform all kinds of operations on the values contained within, from simple slicing to complex interpolation schemes. In the next sections we will go through the main features of pandas, all of wich will be used extensively in the following chapters of this book.

Dataframes

The best way to understand how pandas works is to create a simple **dataframe** and use it to test the functionalities of the library.

Creating a dataframe

We create a simple 3x3 dataframe called **df** that will be used for the next examples:

```
1  df=pd.DataFrame({"A":[4,2,6], "B":[9,1,8], "C":[5,7,4]},
2                  index=[1,2,3])
3  display(df)
```

	A	B	C
1	4	9	5
2	2	1	7
3	6	8	4

The function `pd.DataFrame()` allows the user to initialize a dataframe column by column. By looking at the output of the previous code cell, it is easy to understand how the syntax works: a dictionary (the part contained in the curly brackets) is used to define the various columns, while the optional parameter **index** is used to specify the label assigned at each row. Note that the length of **index** must be equal to the length of the columns of data.

Creating a dataframe from a NumPy array

This is a very common operation, and the syntax is pretty straight forward:

```
1  import numpy as np
2  mat=np.random.randint(0, 10, (4,4))
3  numpy_df=pd.DataFrame(mat, columns=['A', 'B', 'C', 'D'])
4  numpy_df
```

	A	B	C	D
0	2	5	0	7
1	4	6	8	9
2	4	0	7	5
3	3	0	7	0

In the first line we import numpy, since so far we had only imported pandas. Then we create an array of random integer numbers from 0 to 10 with shape (4,4) called **mat**. In line 3 we create the dataframe **numpy_df** using the values stored in **mat**, specifying the column names. This is not mandatory: without the *colums* argument pandas would have created a dataframe with column names ranging from 0 to 3.

Reshaping the data of a dataframe

Pandas offers numerous functions to manage the rows and columns of a dataframe, so let's take a look at those that are used most often.

sort_values

This function is used to sort the rows in a specific order. For example, using the dataframe **df** created previously:

```
1  df.sort_values("B")
```

	A	B	C
2	2	1	7
3	6	8	4
1	4	9	5

As you can see, the rows have been sorted in ascending order with respect to the values contained in the column **B**. Notice that the index of the dataframe has been affected as well, and the indices remain "connected" to their corrispective row.

> **NOTE**
>
> It's important to specify that with functions like *sort_values* the new configuration of the dataframe is not **saved**. In order to maintain the new configuration you must call the function *and* assign the result to the dataframe, like so:
> ```
> df=df.sort_values("B")
> ```

rename

with *rename* you can assign new names to the rows and columns of the dataframe:

```
1  df.rename(columns={"A": "col. 1", "B": "col. 2", "C":"col. 3"},
2            index={1:"row 1", 2:"row 2", 3:"row 3"})
```

	col. 1	col. 2	col. 3
row 1	4	9	5
row 2	2	1	7
row 3	6	8	4

drop

you can use `drop()` to remove columns and rows from a dataframe:

```
1  df.drop(columns={"B"}, index={1,3})
```

	A	C
1	4	5
3	6	4

Sometimes, after deleting a row from a dataframe, you might want to reset the index so that each row is named with a number, starting from zero. This is because deleting a row in the middle of a dataframe will cause the index to become discontinuous. You can solve this problem by calling `reset_index()`:

```
1  df=pd.DataFrame({"A":[2,1,5], "B":[3,5,6], "C":[3,6,8]})
2  df=df.drop(1)
3  display(df)
4  df=df.reset_index(drop=True)
5  df
```

	A	B	C
0	2	3	3
2	5	6	8

in line 2 we use drop to delete the second line of the dataframe. Notice how deleting rows simply requires to pass `drop()` the position of the row you wish to remove. However, after doing this the index is no longer a series of subsequent integers. To fix this we call **reset_index**, specifying that we don't wish to keep the previous index as a column by setting the argument **drop** to *True*.

concat

If you have two dataframes that you want to concatenate together, you can use *concat*:

```
1  df1=pd.DataFrame({"A":[4,2], "B":[9,1]},
2                   index=["row 1","row 2"])
3  df2=pd.DataFrame({"A":[5,3], "B":[7,9]},
4                   index=["row 3","row 4"])
5  res=pd.concat([df1, df2])
6  res
```

	A	B
row 1	4	9
row 2	2	1
row 3	5	7
row 4	3	9

By default, *concat* concatenates dataframes along **axis 0**, stacking them vertically. Notice that in the example above both dataframes have the same columns **A** and **B**. This ensures that the columns are going to stack properly: if you try to concatenate two dataframes with different columns pandas will raise an error.

To stack dataframes horizontally, simply add the argument *axis=1* to the function call. In this case the dataframes must have the same index instead.

Extracting data from a dataframe

A common operation performed on dataframes is **slicing**, something that we have already encountered while discussing numpy arrays.The concept is the same, but the syntax is a little bit different. Let'create a random dataframe and test some of the functionalities that pandas offers for this task:

```
1 df=pd.DataFrame(np.random.randint(0,10, (3,5)),
2                        columns=["a", "b", "c", "d", "e"])
3 df
```

	a	b	c	d	e
0	4	2	2	0	4
1	3	5	7	9	5
2	9	4	9	5	9

Now let's see how to select specific columns from a dataframe.

```
1  display(df[["a", "c"]])
2  display(df.filter(regex="b"))
```

	a	c
0	4	2
1	3	7
2	9	9

	b
0	2
1	5
2	4

To select some of the columns of a dataframe, simply pass a list containing their names, as we see in line 1. To select the columns that match a certain expression, use the *filter* function. the **regex** argument stands for "regular expression", which means that pandas is going to search the columns whose name match the expression specified by **regex**. In the example above, we ask to select the column named *b*. This is pretty basic, however using **regex** you can specify much more complex criterions. Here are some examples:

- **"\,":** Finds strings containing a comma ","
- **"size$":** Finds strings ending with the word "size"
- **"^size":** Finds strings beginning with the word "size"
- **"^b[1-3]$":** Finds strings beginning with "b" and ending with 1,2,3

These two commands are used to slice both rows and columns of a dataframe. The syntax is pretty similar to what we saw for numpy arrays, as you can see from the next code cell:

```
1  display(df.loc[:, "b":"d"])
2  display(df.iloc[2, 3]
```

	b	c	d
0	2	2	0
1	5	7	9
2	4	9	5

5

The difference between *loc* and *iloc* is that with *loc* you must use the names of the columns and the rows, while with *iloc* you basically treat the dataframe like a NumPy array specifying the rows and columns with numbers. In the first line we tell pandas that we want to extract data from every row (using `:`) and from the columns ranging from **b** to **d** (using `"b":"d"`). In the second line we use *iloc* select the element in position `(2,3)` in the dataframe.

> **NOTE**
>
> Slicing with *loc* returns a pandas dataframe, but slicing with *iloc* returns a number.

Creating a dataframe with complex headers

Sometimes it is useful to structure data in a more comprehensive way other than simple rows and columns. This is where the **MultiIndex** function comes into play. In the next code snippet we create a 4x4 dataframe with a nested header and index that we will use in later examples.

```
1  header=pd.MultiIndex.from_tuples([("A", "x"), ("A", "y"),
2                                    ("B", "u"), ("B", "v")])
3  ind=[np.array(["M", "M","N", "N"]),
4       np.array(["bar", "foo", "baz", "qux"])]
5  data=np.array([[9,4,8,5],[4,4,0,1],[5,7,4,5],[6,6,2,2]])
6  df=pd.DataFrame(data, columns=header, index=ind)
7  df
```

		A		B	
		x	y	u	v
M	bar	9	4	8	5
	foo	4	4	0	1
N	baz	5	7	4	5
	qux	6	6	2	2

As you can see, before we create the dataframe we must first create the **header** and **index**. We use the function *pd.MultiIndex.from_tuples* to build the header, passing a list of tuples that defines the hierarchy of the columns. The first element of the tuples creates the first level of the dataframe, and so on. Then in line 3 we create the index for the rows using a list of NumPy arrays. The first array defines the first level of the index, and so on. In line 5 we create an array for the data, making sure that the dimensions match the header and index that were created before. Finally we build the dataframe in line 6, passing **header** and **ind** to the *column* and *index* arguments.

To slice a multi-index dataframe you can use *loc* or *iloc*. With *loc*, you can specify the sub-columns and the sub-rows using tuples:

```
1  df.loc[("M", "foo"),("A", "x")]
```

```
4
```

Performing operations with dataframes

Something that often comes up when using dataframes are algebraic operation between columns, for example multiplying two columns and storing the result in a new one. In the next code cell we calculate the second moment of inertia of a rectangle, given the width **b** and he height **h**.

```
1  b=np.array([1,1,1])
2  h=np.array([1,2,3])
3  df=pd.DataFrame({"b":b, "h":h})
4
5  df["I"]=(df.b*(df.h)**3)/12
6  df
```

	b	h	I
0	1	1	0.083333
1	1	2	0.666667
2	1	3	2.250000

In lines 1 and 2 we create two arrays that contain the dimensions of the rectangles. Then in line 3 we build the dataframe from these two arrays, and finally in line 4 we compute the second moment of inertia using the formula

$$\frac{b \cdot h^3}{12}$$

The result is stored in a new column called **I**.

I suggest experimenting a bit with some random dataframes before moving on to the next sections, trying the various functions explained above. Now we will learn how to import data from external sources.

Working with external data

More often than not engineers must perform operations using data stored in *.csv* or *.xlsx* files. Pandas has built-in functions to load these types of files and store them automatically in dataframes, and also handle missing data.

loading a spreadsheet

For the next example you will need the *column_example.xlsx* file that you can find on the book's support website at https://python4civil.weebly.com/pandas.html. Once you have downloaded the file, put it in the same folder as the jupyter file you are using. If the file isn't in the same folder pandas won't know where to find it and won't be able to load it. Once you have everything ready, you can load the file in jupyter with the following code:

```
1  df=pd.read_excel('column_example.xlsx', header=[0,1], index_col=0)
2  df=df.round(4)
3  df.head()
```

Column						
x	M neg	M pos	V neg	V pos	N neg	N pos
0	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.0000
1	0.0000	0.0	2.9919	0.0	0.5679	0.0	806.8224
2	0.1675	0.0	2.9024	0.0	0.5679	0.0	806.8224
3	0.3350	0.0	2.8133	0.0	0.5678	0.0	806.8224
4	0.5025	0.0	2.7247	0.0	0.5677	0.0	806.8224

The file we are importing contains the bending moment, shear and axial force of a column part of a concrete frame. If you take a look at the *.xlsx* file you will notice that the data has a header that says **column** in the first line and **x**, **M neg**, **M pos**,**T neg**, **T pos**, **N neg**, **N pos** in the second. The first column represents the position (height) considered.

To import the data we use the function *read_excel*, specifying the name of the file we want to load. We also pass the **header** argument, specifying which lines will be used as headers for the dataframe. We could have specified an index as well by passing the **index_col** argument, telling pandas which column must be used as index. In line two we round the values so that they have maximum four decimal digits, and finally using `head()` we display the first few lines of the dataframe.

> **NOTE**
>
> When importing data in a dataframe, it's important to have at least a general idea of how the data is structured. So take some time to examine the file you are loading and if possible organize the data so that it's easy to manipulate. This will save a lot of time down the line cleaning an ill-presented dataframe.

The dataframe we created still needs some work. For example, we really don't need the first level of the header that says "column". We also don't need the first line of data, since it's composed of zeros. Plus, if we take a look at the last rows of the dataframe we see that there is a line with missing data. To display the last rows use the **tail()** function:

```
1  df.tail()
```

	Column						
	x	M neg	M pos	V neg	V pos	N neg	N pos
21	3.350	0.0	1.3601	0.0	0.5140	0.0	806.8227
22	3.350	0.0	3.0870	0.0	0.6027	0.0	220.7086
23	5.025	0.0	2.1503	0.0	0.6027	0.0	220.7086
24	6.700	0.0	1.3178	0.0	0.6027	0.0	220.7086
25	6.700	NaN	NaN	NaN	NaN	NaN	NaN

Making these modifications is really simple and only takes a few lines of code:

```
1  df.columns=df.columns.droplevel(0)
2  df=df.dropna()
3  df=df.drop(0)
4  df.head()
```

	x	M neg	M pos	V neg	V pos	N neg	N pos
1	0.0000	0.0	2.9919	0.0	0.5679	0.0	806.8224
2	0.1675	0.0	2.9024	0.0	0.5679	0.0	806.8224
3	0.3350	0.0	2.8133	0.0	0.5678	0.0	806.8224
4	0.5025	0.0	2.7247	0.0	0.5677	0.0	806.8224
5	0.6700	0.0	2.6365	0.0	0.5673	0.0	806.8224

In the first line we delete the first level of the header using **droplevel**, then in line 2 we use the function **dropna** to delete every row that contains a *NaN* value, and finally we delete the first line of the dataframe (the one that contains only zero values) with the **drop** function. At last our dataframe is ready and can be used for further calculations.

Loading a csv file

Comma Separated Values (csv) files are another common format used to store data. You can open a csv file with any text editor that you have installed in your computer. Download the file *beam_example.csv* from https://python4civil.weebly.com/pandas.html, and place it in the same folder as the jupyter notebook you are currently working on. The data in the file is similar to what we have seen previously in **column_example.xlsx**. To load the file we use **read_csv**:

```
1  df=pd.read_csv('beam_example.csv', header=[1])
2  df=df.round(3)
3  df=df.dropna()
4  df=df.reset_index(drop=True)
5  df=df.drop(0)
6  df=df.reset_index(drop=True)
7  df.head()
```

	x	M neg	M pos	T neg	T pos	N neg	N pos
0	0.000	-32.145	0.000	57.936	0.0	-0.073	0.073
1	0.069	-28.255	0.000	55.435	0.0	-0.073	0.073
2	0.137	-24.536	0.000	52.935	0.0	-0.070	0.070
3	0.206	-20.988	0.000	50.435	0.0	-0.067	0.067
4	0.275	-17.613	0.626	47.934	0.0	-0.064	0.064

Notice how we specify that we want to use the second line of the dataframe as header, then we round all the number so that they have a maximum of three decimal places. In line 3 we call **dropna()** to clean the data, and then we reset the index using **reset_index**. In line 5 we delete the first line of data as well, and then we reset the index once again.

> **NOTE**
>
> The function **read_csv** offers a lot of parameters to import the data in the best possible way. For example if you have a csv file that use a period instead of a comma as the separator, you can pass the **sep** argument and specify a different string. Please refer to the documentation for more in-depth information about this command.

Exporting a dataframe

Saving a dataframe in a *.csv* or a *.xlsx* file is very easy, simply call `to_excel()` or `to_csv()`:

```
1  df.to_excel("output.xlsx")
2  df.to_csv("output.csv")
```

If you open up the folder where the jupyter notebook you are working on is stored, you will see that python has created two new files called *output.xlsx* and *output.csv*.

EXAMPLE: load cases for a two span beam

The goal of this exercise is to apply to a practical situation what we have learned up until now about **NumPy**, **SymPy** and **Pandas**. In the end we will have an useful piece of code that can be adapted to a lot different situations.

Problem definition

We have a two-span continuous beam, and we want to consider the two different load combinations shown in the next figure. We want to build a DataFrame that contains the shear and the bending moment for both combinations, and also plot the results.
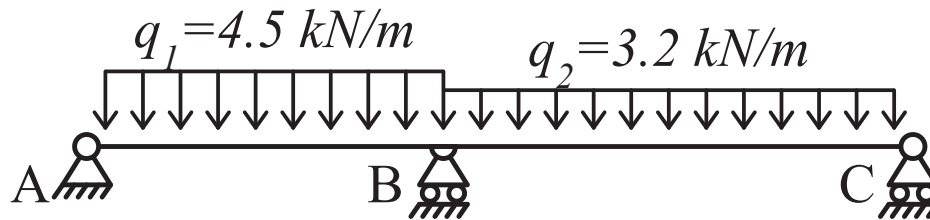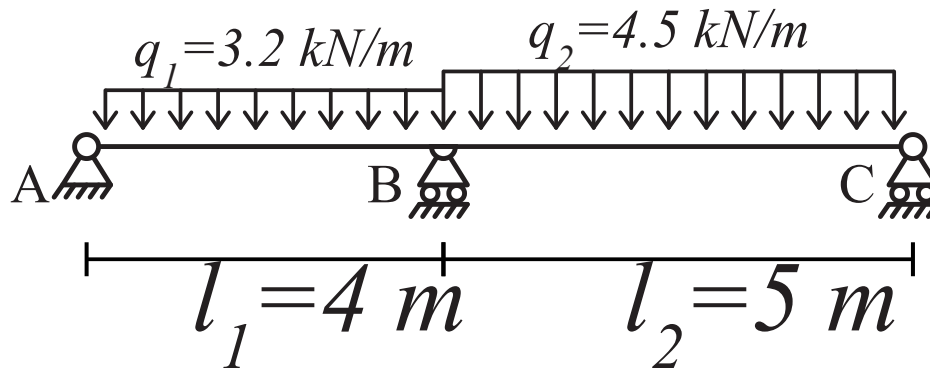
The best approach to solve this problem is to write a **function** to wich we pass the beam's dimensions and loads that returns the bending moment and shear. Since the beam is hyperstatic, to find the reactions we can't simply solve the equilibrium equations. So we split the beam in two and introduce an unknown bending moment M_x to mantain congruence, as shown in the next figure:
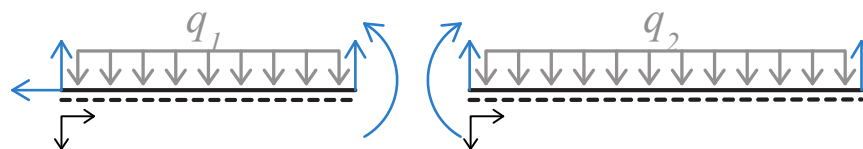
On the left and right of **B** the rotation must be the same, so we can calculate M_x by solving

$$\frac{M_x l_1}{3} + \frac{q_1 l_1^3}{24} + \frac{M_x l_2}{3} + \frac{q_2 l_2^3}{24} = 0$$

Once we know the value of M_x the next step is to calculate the reactions by solving two systems of equations, one for each span:

$$\begin{cases} H_A = 0 \\ V_A + V_{B1} - q_1 l_1 = 0 \\ V_{B1} l_1 - \dfrac{q_1 l_1^2}{2} + M_x = 0 \end{cases} \qquad \begin{cases} V_{B2} + V_C - q_2 l_2 = 0 \\ V_{B2} l_2 - \dfrac{q_2 l_2^2}{2} + M_x = 0 \end{cases}$$

Then we can calculate the bending moment and shear. For the first span:

$$M_1 = V_A x_1 - \frac{q_1 x_1^2}{2}$$

$$V_1 = V_A - q_1 x_1$$

For the second span:

$$M_2 = M_x - \frac{q_2 x_2^2}{2} + V_{B2} x_2$$

$$V_2 = V_{B2} - q_2 x_2$$

Let's implement all of this in **python**. The function that calculates the internal forces is displayed in the next code cell:

```python
1   import numpy as np
2   import sympy as sp
3   import pandas as pd
4
5   def solve_beam(l1, l2, q1, q2):
6       l=l1+l2 #total length
7       Mx=sp.symbols('Mx') #create symbol Mx
8
9       #calculate Mx
10      Mx=sp.solveset(Mx*l1/3+q1*l1**3/24+Mx*l2/3+q2*l2**3/24,Mx).args[0]
11
12      #sove equilibrium equations
13      Va, Vb1, Vb2, Vc=sp.symbols('Va Vb1 Vb2 Vc')
14      Va, Vb1=sp.linsolve([Va+Vb1-q1*l1, Vb1*l1+Mx-(q1*l1**2)/2],
15                                           (Va, Vb1)).args[0]
16      Vc, Vb2=sp.linsolve([Vb2+Vc-q2*l2, Vb2*l2+Mx-(q2*l2**2)/2],
17                                           (Vc, Vb2)).args[0]
18      Vb=Vb1+Vb2
19
20      x1=np.arange(0,l1+0.1,0.1) #create axis x1
21      x2=np.arange(0, l2+0.1, 0.1) #create axis x2
22
23      beam1=pd.DataFrame({"x":x1}) #create a dataframe for the first span
24      beam2=pd.DataFrame({"x":x2}) #create a dataframe for the second span
25
26      beam1["M"]=Va*beam1.x-(q1*beam1.x**2)/2 # calculate M and store it
27      beam2["M"]=Mx-(q2*beam2.x**2)/2+Vb2*beam2.x # calculate M and store it
28
29      beam1["V"]=Va-q1*beam1.x # calculate V and store it
30      beam2["V"]=Vb2-q2*beam2.x # calculate V and store it
31
32      beam2.x=beam2.x+l1 # re-assign x for the second span
33
34      beam=pd.concat([beam1, beam2]) # concatenate the two dataframes
35
36      return(beam) # return the result
```

The code looks complicated, but in reality it's quite simple. Let's explain what each line does.

- **Line 1 to 3:** we import the necessary libraries
- **Line 5:** this is the function's definition. We create a function called **solve_beam** that accepts **l1**, **l2**, **q1**, **q2** as inputs.
- **Line 6:** we calculate the total length of the beam.
- **Line 7:** we create a symbol for the unknown moment M_x.
- **Line 10:** using *solveset*, we solve the congruence equation for **Mx**, and extract the result using **args[0]**. We store the result in **Mx**, wich now is no longer a SymPy symbol but a normal *float* variable.
- **Line 13:** we create the symbols **Va**, **Vb1**, **Vb2** and **Vc**
- **Lines 14 to 17:** we call *linsolve* to solve the systems for the two spans. Like we did with **Mx**, we store the results in the corresponding variables

- **Lines 20 and 21:** we initialize the two numpy arrays that will define the x coordinate for the two spans. We use the numpy function *arange*, that takes as input a starting value, an ending value and a step. In this case we want to discretize the two spans with a set of points with step equal to 0.1 m. The results looks like this: `x1=[0.1, 0.2, 0.3, ..., l1]`
- **Line 23 and 24:** We are treating the two spans separately, so we create two separate dataframes. We store the two x coordinate that we have just created in column **x**
- **Line 26 and 27:** Using the formulas explained in the previous section we calculate the bending moment for each span. Notice how we use the **x** of the dataframes to create a new **M** column with the same length.
- **Line 29 and 30:** We create calculate **V** for each dataframe and we store the result in a new column.
- **Line 32:** We have implemented all the formulas that we need. The next step would be to concatenate the two dataframes we just created in order to obtain a single dataframe for the whole beam. However, the **x** coordinate contained in **beam2** starts from 0. This was useful for calculating the bending moment and shear, but now we want it to start from **l1** so that when we concatenate the two dataframes tha column **x** goes from 0 to `l`. In order to do this, we simply add **l1** to `beam2.x`.
- **Line 34:** here we concatenate the two dataframes using *concat*
- **Line 36:** finally, we specify that the function must return the dataframe **beam**.

Well, that wasn't that difficult: we simply implemented the formulas that we had stated previously. In the next code cell we use the function that we have just created to solve the beam for the two load cases.

```
 1  header=pd.MultiIndex.from_tuples([("combo 1", "M"), ("combo 1", "V"),
 2                                     ("combo 2", "M"), ("combo 2", "V")])
 3  combos=pd.DataFrame(columns=header)
 4  combos["x"]=solve_beam(4, 5, 3.2, 4.5)["x"]
 5
 6  combos["combo 1"]=solve_beam(4, 5, 3.2, 4.5)
 7  combos["combo 2"]=solve_beam(4, 5, 4.5, 3.2)
 8  combos=combos.set_index("x")
 9
10  combos=combos.astype("float")
11  combos.head()
```

x	combo 1		combo 2	
	M	V	M	V
0.0	0.000000	3.735764	0.000000	6.611111
0.1	0.357576	3.415764	0.638611	6.161111
0.2	0.683153	3.095764	1.232222	5.711111
0.3	0.976729	2.775764	1.780833	5.261111
0.4	1.238306	2.455764	2.284444	4.811111

let's go through each line like we did previously:

- **Line 1:** here we create the header for the dataframe containing the load combinations.
- **Line 3:** we build an empty dataframe called **combos** using the header just created.
- **Line 4:** In the dataframe we just created, we create a new column that contains the **x** coordinate of the beam. the most simple way to do this is to just call **solve_beam** and extract the **x** column using ["x"].
- **Line 6 and 7:** here we fill the columns of **combos** by calling *solve_beam* twice and passing the loads for each combination.
- **Line 8:** instead of having the default index, we want the dataframe to be indexed using **x**, so we call *set_index* to tell pandas wich column we want to use as index.
- **Line 10:** This line is a bit problematic. You might think that the values stored in **combos** are floats, but actually they are some different kind of datatype that pandas uses. You can print the datatype of **combos** using `print(combos.dtypes)`. This will display the datatype of each column, and they will all say *object*. If we keep the dataframe like this, the plotting functions that we will use later wont' work. To fix this we call the function *astype* and specify "float". This will convert all the number in the dataframe to *float*.
- **Line 11:** displays the first rows of the dataframe.

Plotting the results

The library *matplotlib* (used for plotting) will be explained in the next chapter. For now just copy the following code in an empty cell and run it:

```python
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,8))
ax = plt.subplot(211)
ax.invert_yaxis()

combos.loc[:,pd.IndexSlice[:,"M"]].plot(ax=ax)

ax = plt.subplot(212)
ax.invert_yaxis()
combos.loc[:,pd.IndexSlice[:,"V"]].plot(ax=ax)
```
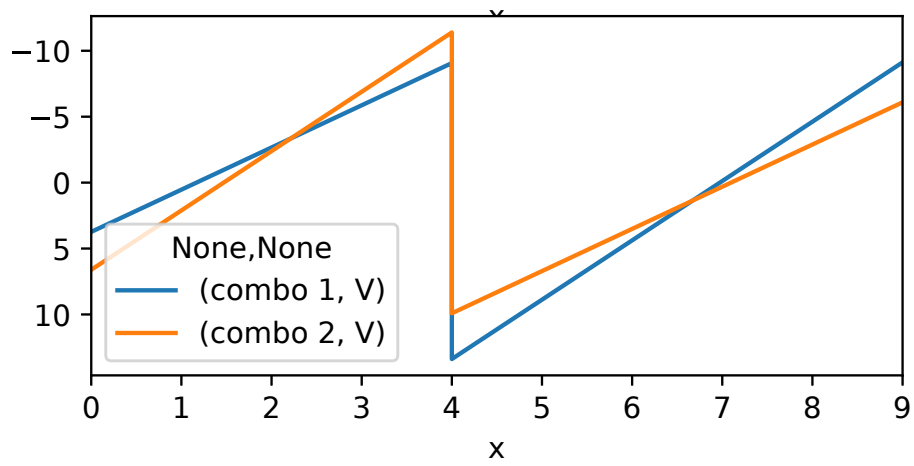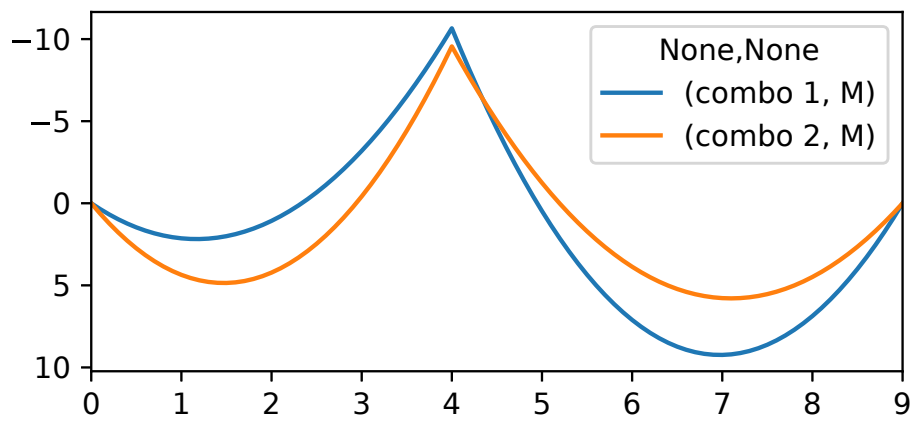
Matplotlib

In this chapter you will learn how to visualize various kinds of data using the functionalities offered by **Matplotlib**. Those who have used *MatLab* at some point in their lives will find this library somewhat familiar. Indeed Matplotlib was written to mimic the behaviour of *Matlab*, at least to some extent. In the vast landscape of python libraries matplotlib as become virtually the *only* library that people use to plot data. Like all the libraries we have seen so far matplotlib is in a sense the "industry standard" to do this kind of task.

In this chapter you will find many code examples that showcase some of the functionalities of matplotlib. The main goal is to give you the tools necessary so that you know what to do in order to obtain a certain type of plot. In the end, however, everything comes down to personal taste. In fact I believe that tinkering and experimenting is the best way to learn how matplotlib works.

Loading the library and importing the data

Before we start exploring matplotlib we must first load the library and import some data to plot. Let's start with the usual preamble:

```
1  import numpy as np
2  import pandas as pd
3  import matplotlib.pyplot as plt
```

In this chapter we will need NumPy, pandas and of course **matplotlib**. The module of matplotlib used for plotting is called **pyplot**. Here we load it with the nickname *plt*.

Now let's import the data. For the examples to come we will use the data stored in a file called "beam.csv". You can download such file from https://python4civil.weebly.com/matplotlib.html. Remember to store it in the same folder as the jupyter notebook you are currently working on, otherwise pandas won't know where to find it!

This data is the final product of the example at the end of the chapter about **pandas**, where we calculated the internal forces of a two span beam. Here the results are stored in a *.csv* file with five columns:

- **x:** the x axis of the beam, that will also become the x axis of our plots
- **M1:** the bending moment of the first load combination
- **V1:** the shear of the first load combination
- **M2:** the bending moment of the second load combination
- **V2:** the shear of the second load combination

In the next code cell the data is loaded in a pandas DataFrame, and then the various columns are stored in separate NumPy arrays.

```
1  data=pd.read_csv("beam.csv")
2  x=np.array(data.x)
3  M1=np.array(data.M1)
4  M2=np.array(data.M2)
5  V1=np.array(data.V1)
6  V2=np.array(data.V2)
```

Now that everything has been loaded we can move on to matplotlib.

How Matplotlib works

The library is pretty straight forward. The main function used for plotting is `matplotlib.pyplot.plot()`, to wich you can pass the data you wish to plot. For example:

```
1  plt.plot(x,M1)
2  plt.show()
```
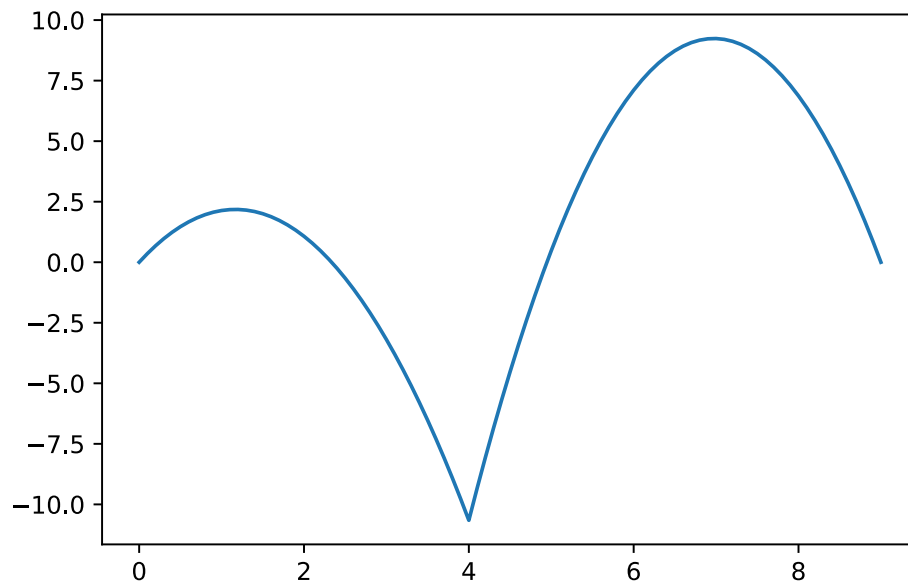

There is obviously a lot of room for improvement. In line 1 we tell matplotlib that we want to plot **M1** using **x** as the horizontal axis, and in line 2 the plot is displayed using `plt.show()` (remember that we assigned the nickname `plt` to `matplotlib.pyplot`). Before we go any further, however, it is best to discuss what are the main components of **matplotlib**.

The graph you see above is composed of two things: a **figure** and an **axes**. The figure is basically the overall frame of the image, and the axes is the actual plot with the ticks, the lines, and so on. "axes" is an instance of matplotlib, so that is why it is referred to as a singular noun. In a single figure you can have multiple axes instances, meaning that you can display more than one plot per figure.

The example above can be replicated using a slightly different syntax:

```
1  fig, ax=plt.subplots()
2  ax.plot(x, M1)
3  plt.show()
```
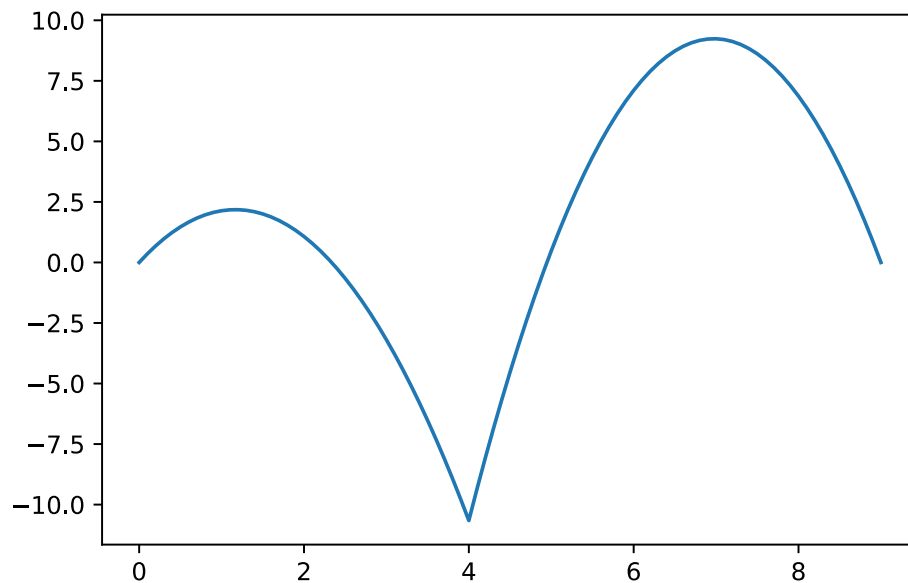
This is a more object-oriented approach. Instead of calling `plot()` directly from **matplotlib.pyplot**, we use the **subplots** command to create a figure and an axes, unpacking them in the two variables **fig** and **ax**. Now we have a figure called **fig** with one single axes instance in it called **ax**. To plot the graph, instead of using **plt** like we did previously, we can use **ax**. At the end we still call `plt.show()` to display the plot.

It is important to know how both approaches work, so that you can understand all the examples you might find online. Now we will add some additional elements to the plot and compare the two methods.

Using **plt**, we can write the following piece of code:

```
1  plt.plot(x,M1, label="M1")
2  plt.xlabel("x")
3  plt.ylabel("bending moment")
4  plt.title("Beam")
5  plt.legend()
6  plt.gca().invert_yaxis()
7  plt.gcf().set_size_inches(6,3)
8  plt.show()
```
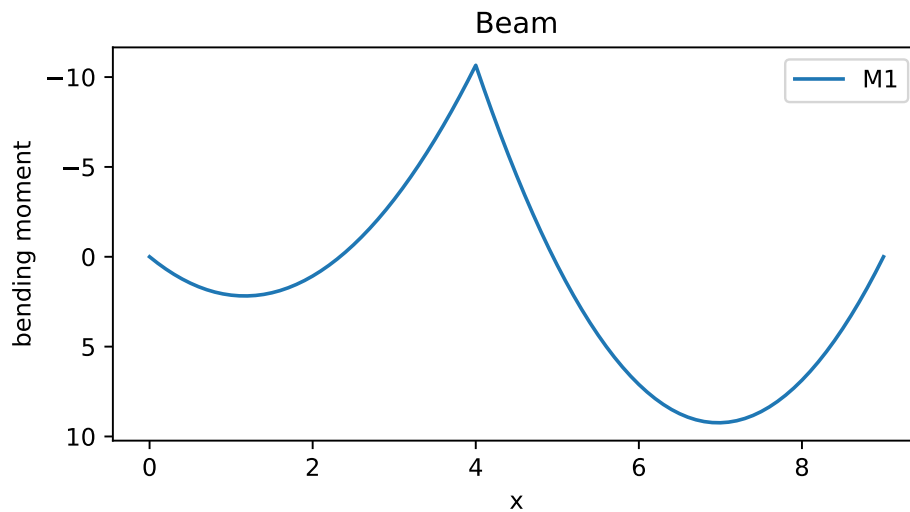
Let's go through the code line by line:

- **Line 1:** Here we invoke **plot** to plot **M1** using **x** as the horizontal axis. We also pass the **label** argument, that specifies the name that will appear in the legend.
- **Line 2:** By using `plt.xlabel` we specify the text that will appear below the horizontal axis
- **Line 3:** This specifies the label for the vertical axis instead
- **Line 4:** here we specify the title of the plot
- **Line 5:** We tell matplotlib that we want to display the legend. Because while calling **plot** we have passed a **label** argument, matplotlib will build the legend automatically.
- **Line 6:** You might have noticed that the bending moment in the output of code was upside-down. This is because the bending moment is usually displayed with positive values below the beam, wich means that to display it correctly we must invert the y axis. In order to do this we must tell the **axes** instance to invert it using `.invert_yaxis()`. To get the current axes instance matplotlib offers the `gca()` function, wich stands for *get current axes*.
- **Line 7:** This line sets the figure size to 6x3 inches. The size is controlled by the **figure** instance, so we have to use `gcf()` (wich stands for *get current figure*) and then use the **set_size_inches** function.
- **Line 8:** Calling `plt.show()` will display the plot.

Now let's plot the same graph using the object-oriented approach:

```
1  fig, ax=plt.subplots()
2  ax.plot(x,M1, label="M1")
3  ax.set_xlabel("x")
4  ax.set_ylabel("bending moment")
5  ax.set_title("Beam")
6  ax.legend()
7  ax.invert_yaxis()
8  fig.set_size_inches(6,3)
9  plt.show()
```
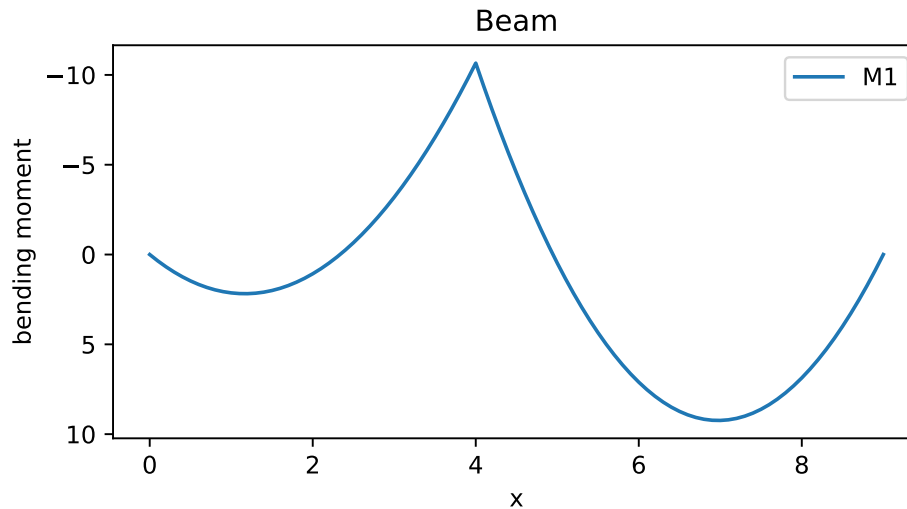

- **Line 1:** Using the **subplots** function we create a figure instance and an axes instance, that get stored in **fig** and **ax**. If you don't specify any arguments inside the parentheses matplotlib will create one single figure with one axes instance inside of it. The variables **fig** and **ax** are assigned by *unpacking* the output of `plt.subplots()`. If you run `print(plt.subplots())` in an empty cell you will see that the function returns a tuple with the **figure** instance in the first position and the **axes** instance in the second position.
- **Line 2:** Here, instead of using **plt** to plot M1, we can use the axes instance we have just created.
- **Line 2 and 3:** Using **ax** we can set the label for the two axes by calling **set_xlabel** and **set_ylabel**. Notice that the syntax is slightly different from the previous example: here **xlabel** has been replaced by **set_xlabel** and **ylabel** has been replaced by **set_xlabel**.
- **Line 4:** To set the title we use **set_title**. You might see a pattern here: when using **ax** the commands usually start with `set_`.
- **Line 6:** we tell **ax** to display the legend
- **Line 7:** Because we have saved the axes instance inside **ax**, we don't need to use `gca()`
- **Line 8:** The same applies for the figure: we already saved the instance of the figure inside **fig**, so There is no need to use `gcf()`

Hopefully now you have an idea of how the two different approaches work. Generally, it is best to avoid using the first method unless you want to plot really simple things. Especially when it is required to plot more than one graph, using **subplots** becomes mandatory. Plus, the code written with the second method is much more readable.

Modifying the appearance of a plot

Matplotlib allows the user to edit almost every single aspect of a plot, and the possibilities are endless. Luckily to obtain a great looking figure you really don't need too much work. In order to keep the code in the following examples short, the lines for setting up the legend, the title etc. have been omitted.

Changing colors and line styles

In the next piece of code we change the appearance of the line:

```
1  fig, ax=plt.subplots(figsize=(8,3))
2  ax.invert_yaxis()
3  ax.plot(x,M1, label="M1", color="deeppink", linestyle="--", linewidth=3)
4  plt.show()
```
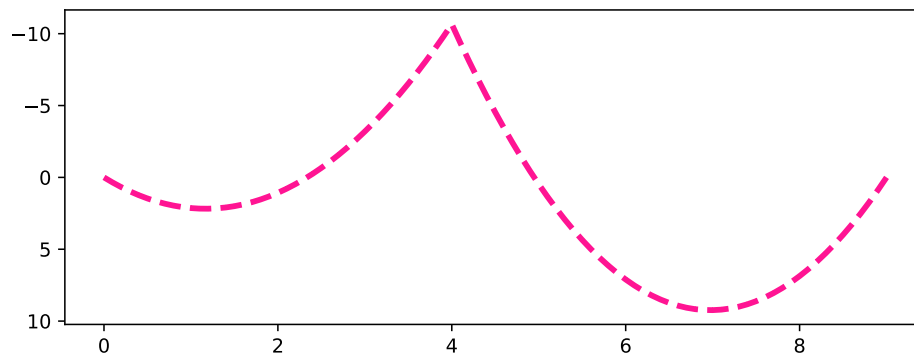

In line 3 we pass **plot** three additional arguments: **color**, **linestyle** and **linewidth**. The line now has a different color, a different width and is also dashed. The value passed to **color** is a string that picks one of the so called "named colors" of matplotlib. You will find a list of all these colors at the end of this chapter. The value passed to **linestyle** is also a string, and again at the end of this chapter you will find all the references you need. **linewidth** accepts a number, and by default is set to 1. Notice how the size of the figure can be set directly when calling `plt.subplots()`, thus saving one line of code.

Plotting more than one line

Multiple datasets can be plotted in the same graph simply by calling **plot** more than once:

```
1  fig, ax=plt.subplots()
2  ax.invert_yaxis()
3  ax.plot(x,M1, label="M1", linewidth=3)
4  ax.plot(x,M2, label="M2", linewidth=3)
5  plt.show()
```
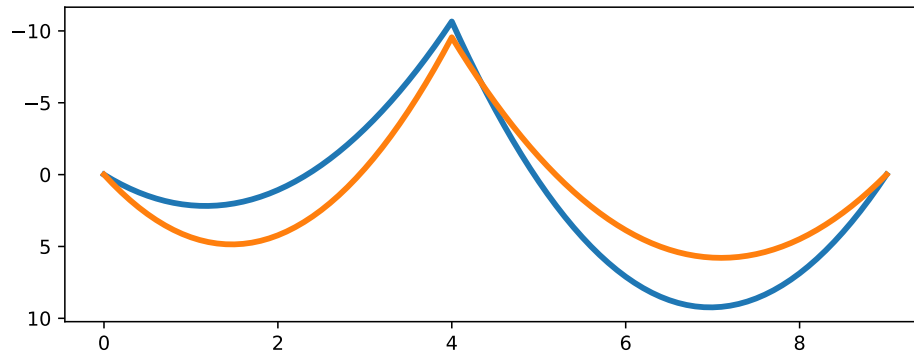
Plotting styles

Matplotlib has buit-in styles that make plotting a professional looking graph really easy. You can set the style by calling `plt.style.use()` before creating the figure:

```
1  plt.style.use("seaborn")
2  fig, ax=plt.subplots(figsize=(8,3))
3  ax.invert_yaxis()
4  ax.plot(x,M1, label="M1", linewidth=3)
5  ax.plot(x,M2, label="M2", linewidth=3)
6  plt.show()
```
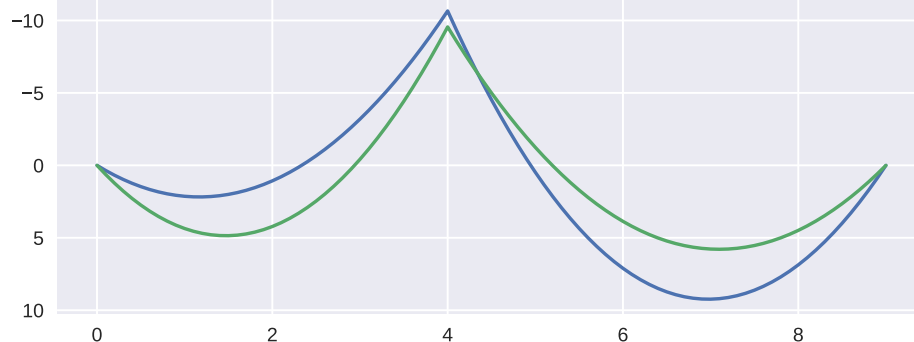
At the end of this chapter you will find a chart containing all the available styles.

Filling areas

With **fill_between** you can fill the area between two curves:

```
1  plt.style.use("seaborn-whitegrid")
2  fig, ax=plt.subplots(figsize=(8,3))
3  ax.invert_yaxis()
4  ax.plot(x,M1, label="M1", linewidth=3)
5  ax.plot(x,M2, label="M2", linewidth=3)
6  ax.fill_between(x, M1, alpha=0.3)
7  ax.fill_between(x, M2, alpha=0.3)
8  plt.show()
```
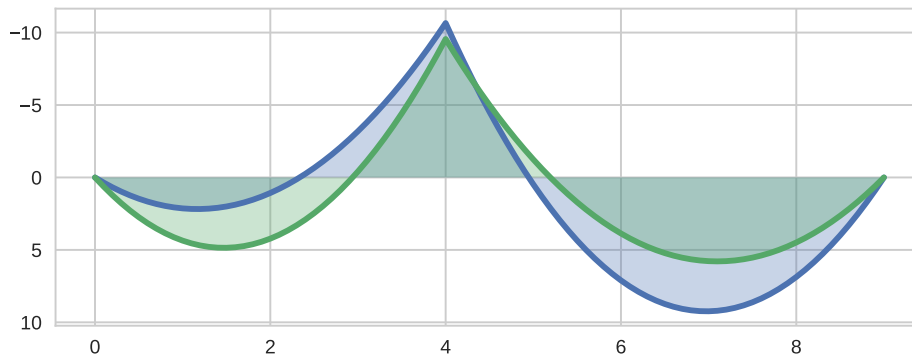

fill_between needs at least the x coordinates of the points and their y value, so we pass **x** and **M1**. The transparency of the fill can be adjusted with the parameter **alpha**. `alpha=0` means completely transparent, `alpha=1` means completely opaque. By specifying another set of y values you can fill the area between two curves:

```
1  plt.style.use("seaborn-whitegrid")
2  fig, ax=plt.subplots(figsize=(8,3))
3  ax.invert_yaxis()
4  ax.plot(x,M1, label="M1", linewidth=3)
5  ax.plot(x,M2, label="M2", linewidth=3)
6  ax.fill_between(x, M1, alpha=0.3)
7  ax.fill_between(x, M2, alpha=0.3)
8  plt.show()
```
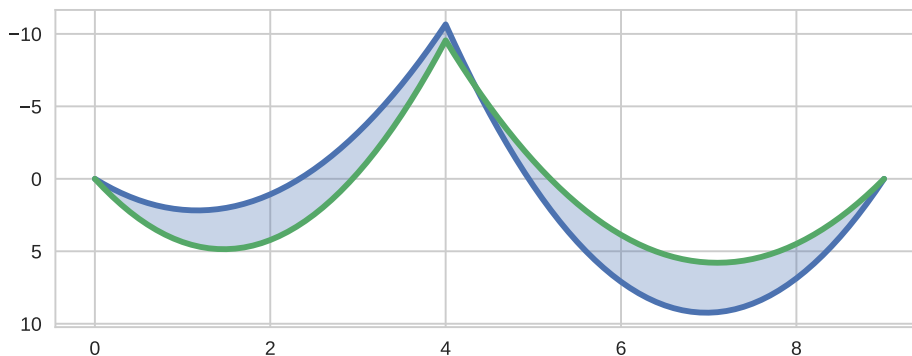

Finally, with the parameter **where** you can tell matplotlib to fill only the portions of the plot that fulfill a certain condition:

```
1  fig, ax=plt.subplots(figsize=(8,3))
2  ax.invert_yaxis()
3  ax.plot(x,M1, label="M1", linewidth=3)
4  ax.fill_between(x, 0, M1, alpha=0.3, where=(M1<=0))
5  ax.fill_between(x, 0, M1, alpha=0.3, where=(M1>=0))
```
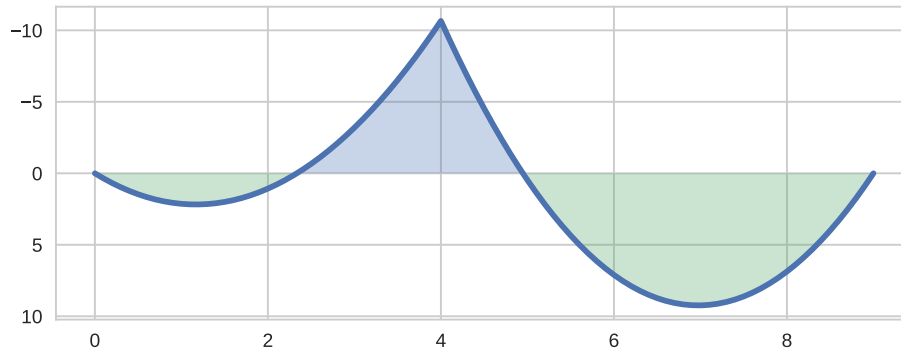

Adding markers

A marker is a symbol that matplotlib displays in correspondence of each data point. Markers are also what constitute scatter plots, as we will see later. When passing the **marker** argument to `plot()`, each datapoint is visualized with a symbol. We can use this to our advantage to display the supports of the beam we are printing:

```
1  fig, ax=plt.subplots(figsize=(8,3))
2  ax.invert_yaxis()
3  ax.plot(x,M1, label="M1")
4  ax.plot(x,M2, label="M2")
5  ax.fill_between(x, 0, M1, alpha=0.3)
6  ax.fill_between(x, 0, M2, alpha=0.3)
7  ax.plot([0,4,9],[0,0,0], linewidth=2, color="dimgrey",
8          marker=6, markersize=16)
9  plt.show()
```
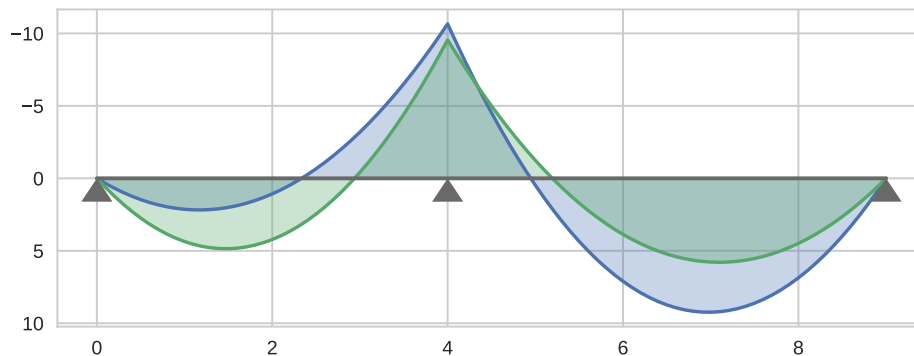

There are a lot of types of markers, please refer to the chart at the end of this chapter to pick the one most suited for you.

Plotting multiple plots

Up until now we have used matplotlib to display only one graph at a time, leaving the parentheses of `plt.subplots()` empty. Before doing anything it is better to understand how matplotlib organizes the various subplots inside a figure. Remember that a "plot" is just an **axes** instance. If you have more than one, matplotlib is going to organize them inside the figure in a grid-like manner. This is why in order to specify how many plots you want you have to pass **subplots** the number of rows and the number of columns of the grid. If there are two rows and two columns, the figure will have a total of four subplots.

```python
 1  fig, (ax1, ax2)=plt.subplots(nrows=2, ncols=1, sharex=True, figsize=(8, 6))
 2  ax1.plot(x,M1, label="M1")
 3  ax1.set_ylabel("bending moment")
 4  ax1.set_title("Combo 1")
 5  ax1.invert_yaxis()
 6
 7  ax2.plot(x,M2, label="M2")
 8  ax2.set_ylabel("bending moment")
 9  ax2.set_title("Combo 2")
10  ax2.invert_yaxis()
11  plt.show()
```
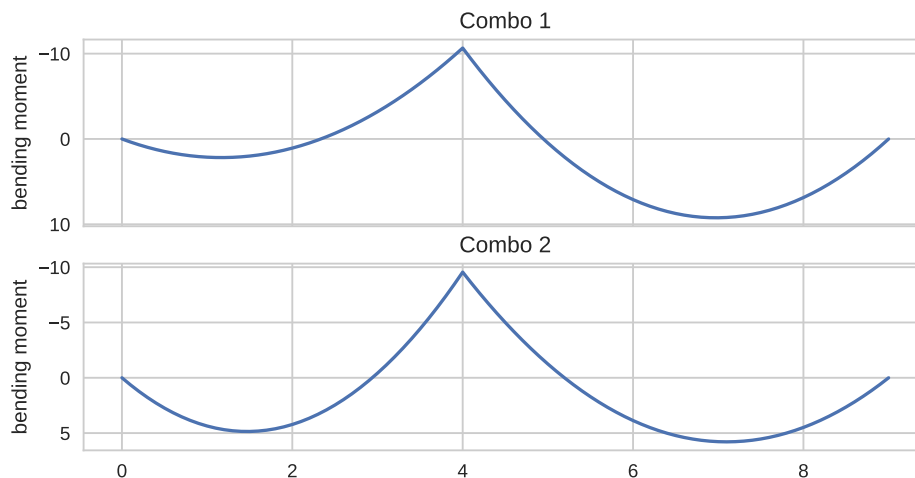

In line 1 we pass `subplots()` the two arguments **nrows** and **ncols**, that define the number of rows and columns in the figure. We also set **sharex** to *True*, meaning that the two plots will share the same x axis. Because there are two plots in the figure, there are also two separate **axes** instances. If you print the output of `plt.subplots` using `print()` you will see that it is structured in this way: `(figure, (axes 1, axes 2))`. When unpacking the output you have to mirror this structure in the variables before the = sign. We now have one figure called **fig** and two axes called **ax1** and **ax2**. The rest of the code is nothing new, except we can now choose in wich plot we want to display the data using **ax1** and **ax2**.

A figure with four plots works this way instead:

```
 1  fig, ((ax1, ax2),(ax3,ax4))=plt.subplots(nrows=2, ncols=2, sharex=True)
 2  fig.set_size_inches(8, 4)
 3  ax1.plot(x,M1, label="M1")
 4  ax1.set_ylabel("bending moment")
 5  ax1.set_title("Combo 1")
 6  ax1.invert_yaxis()
 7
 8  ax2.plot(x,V1, label="V1")
 9  ax2.set_title("Combo 1")
10
11  ax3.plot(x,M2, label="M2")
12  ax3.set_ylabel("bending moment")
13  ax3.set_title("Combo 2")
14  ax3.invert_yaxis()
15
16  ax4.plot(x,V2, label="V2")
17  ax4.set_title("Combo 2")
18  plt.show()
```
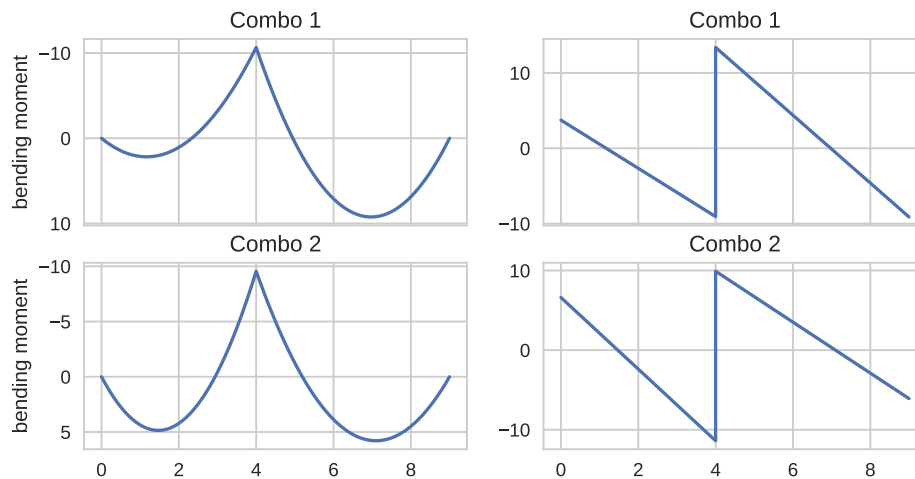

Notice that the structure of the output of `plt.subplots()` has changed: now the **axes** instances are organized inside a 2x2 tuple.

Modifying the tick marks

Sometimes the default numbering that matplotlib uses for the axes does not represent the data very well. To set your own ticks you can use **set_xticks** and **set_yticks**. In the following example we tell matplotlib to display the x position every meter:

```
1  fig, ax=plt.subplots(figsize=(8,3))
2  ax.invert_yaxis()
3  ax.plot(x,M1, label="M1")
4  ax.set_xticks(np.arange(0,10,1))
5  plt.show()
```
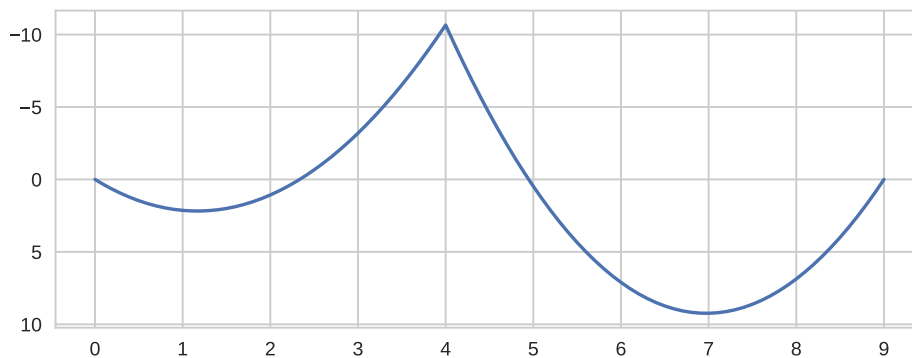

we simply pass `ax.set_xticks()` an array of number ranging from 0 to 9 using `np.arange(0,9,1)`.

Scatter plots

Another very popular style of plots are **scatter plots**. They are mostly used to show correlation between data and have a wide range of applications. We will use a new dataset called *raindata.csv*, that stores informations about precipitations in Italy. The first column contains the name of all the italian provinces, the second the average temperature, the third the average daily precipitations in mm, and the fourth the total number of days in which precipitations exceeded 10 mm in the period from 2007 to 2017. In the next code cell we store all this data in NumPy arrays:

```
1  raindata=pd.read_csv("raindata.csv")
2  province=np.array(raindata["province"])
3  T=np.array(raindata["T"])
4  daily=np.array(raindata["daily prec."])
5  ndays=np.array(raindata["days exceeding 10 mm 2007-2016"])
```

The function of matplotlib that deals with scatter plot is called **scatter**. In the next code cell we use it to see if there is a correlation between the average daily precipitations and the number of days in which the precipitations exceeded 10 mm.

```
1  plt.style.use("seaborn-whitegrid")
2  fig, ax=plt.subplots()
3  ax.scatter(daily,ndays, c=T, cmap="Blues", marker="p")
4  ax.set_xlabel("daily prec.")
5  ax.set_ylabel("days exceeding 10 mm")
6  plt.show()
```
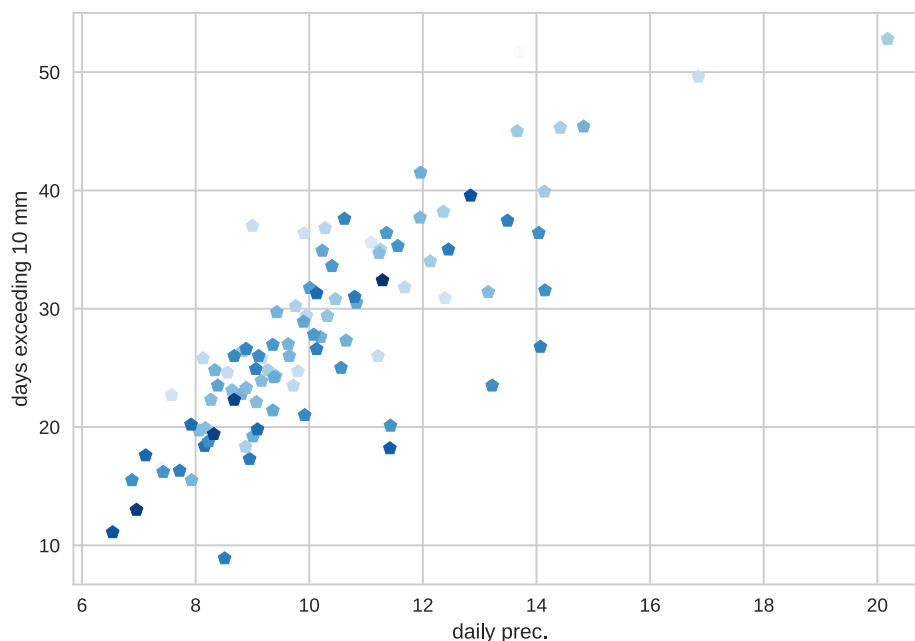
We create a **figure** and an **axes** with the usual preamble, then we call `ax.scatter()`. The arguments you can pass to `ax.scatter()` are slightly different from those of `ax.plot()`. We can actually color-code the points by passing an array of values to the argument **c**: in this case we want the points to display a different color based on the average daily temperature, so we pass **T**. The **cmap** argument defines which color map will be used to display the color, in this case we choose `"Blues"`. At the end of this chapter you will find a chart that lists all the available color maps. We also change the marker used to display the data to a pentagon, by passing `"p"` to the **marker** argument.

To further embelish the graph we can add a **color bar**:

```
1  fig, ax=plt.subplots()
2  im=ax.scatter(daily,ndays, c=T, cmap="Blues", marker="p")
3  ax.set_xlabel("daily prec.")
4  ax.set_ylabel("days exceeding 10 mm")
5  cbar=fig.colorbar(im, ax=ax)
6  cbar.set_label("avg. temperature")
7  plt.show()
```
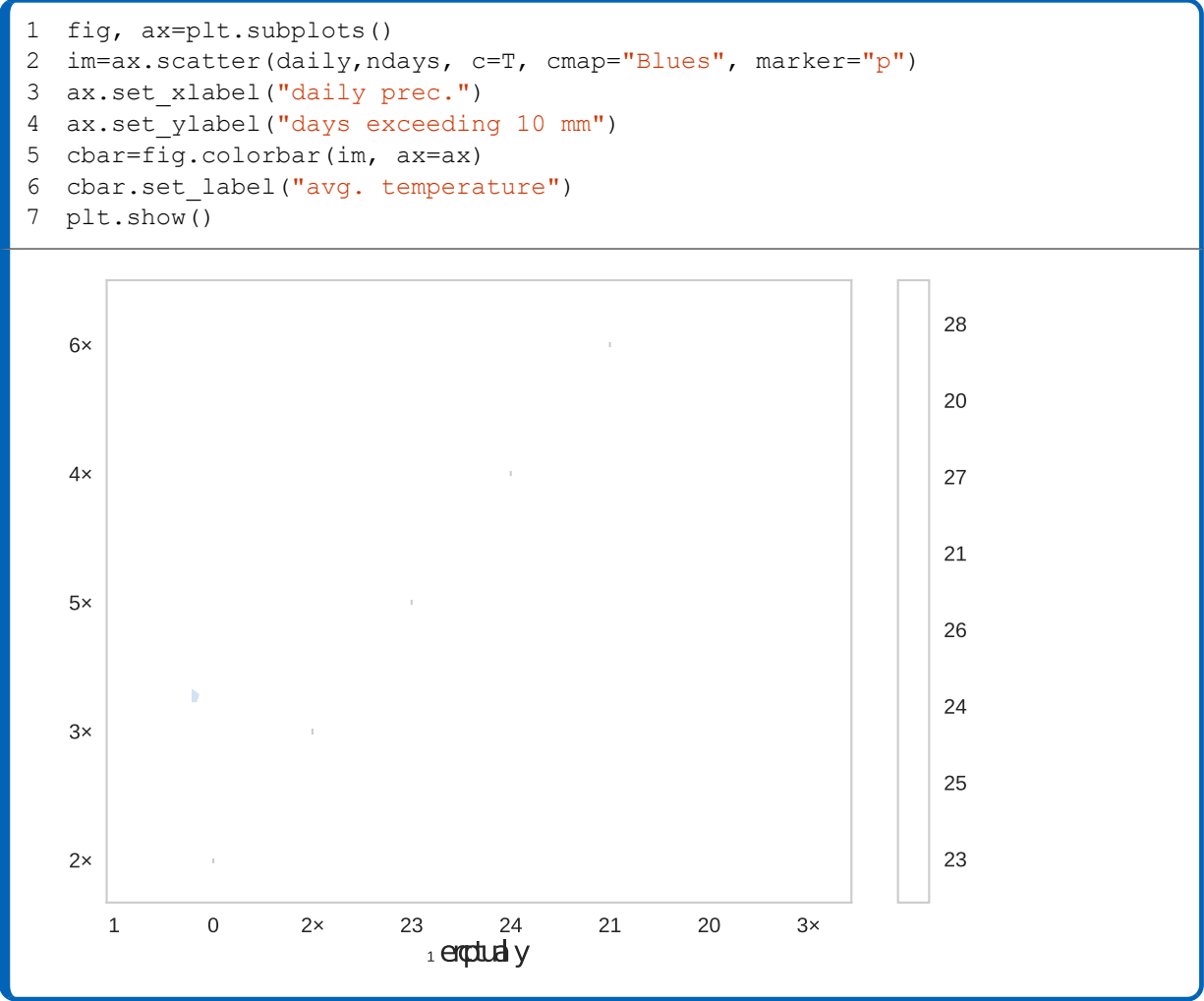

This time when we invoke **scatter** in line 2 we store the output in a variable called **im**. The output of `ax.scatter()` is what matplotlib calls a **PathCollection**. We need this PathCollection because it is the first argument that we have to pass to the function `fig.colorbar()` in line 5. Optionally, you can specify the **axes** instance where you want the colorbar to be displayed by using the **ax** parameter. In our case this doesn't affect the appearance of the plot, since there is only one axes. `fig.colorbar()` creates a colorbar instance that we store inside a variable called **cbar**, which is later used to set the label for the colorbar.

Sometimes to better visualize the data you might need to use a logarithmic scale for the axes. This is done with `plt.xscale("log")` or `ax.set_xscale("log")`:

```python
1  plt.scatter(daily,ndays, c=T, cmap="Blues", marker="p")
2  plt.xscale("log")
3  cbar=plt.colorbar()
4  plt.show()
```
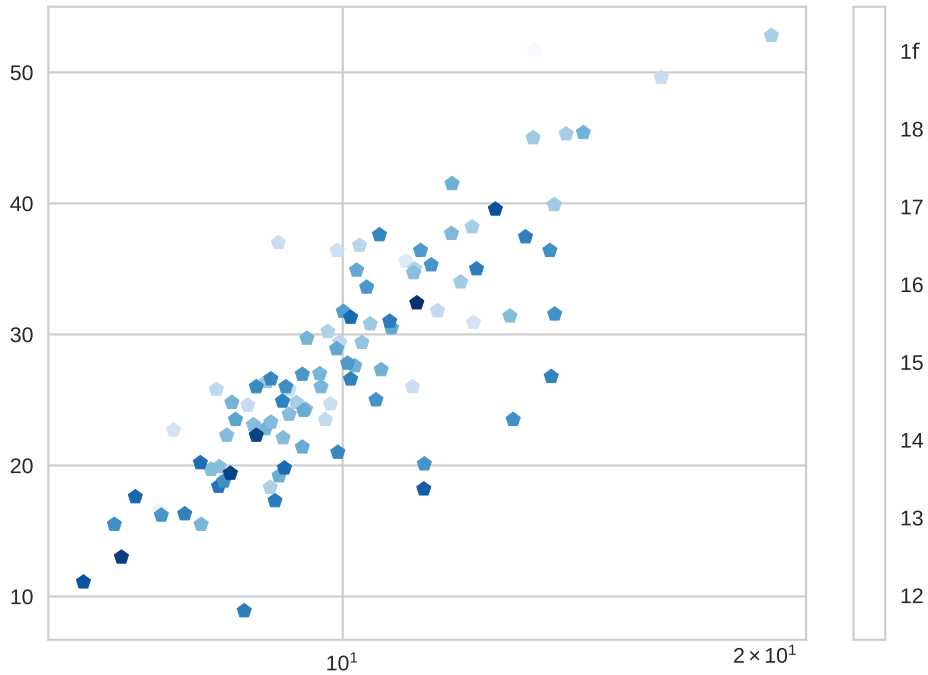

Here instead of initializing **ax** and **fig** we use **plt** directly.

Bar plots

In many situations another meaningful way to represent data is using **bar plots**. For this purpose matplotlib offers the **bar** function. Using the same precipitation data already imported for **scatter** plots, we can display the average temperature for the first fifteen provinces:

```
1  fig, ax = plt.subplots()
2  ax.bar(province[:15], daily[:15])
3  plt.draw()
4  ax.set_xticklabels(ax.get_xticklabels(), rotation=45)
5  ax.set_ylabel("Avg. temperature")
6  plt.show()
```
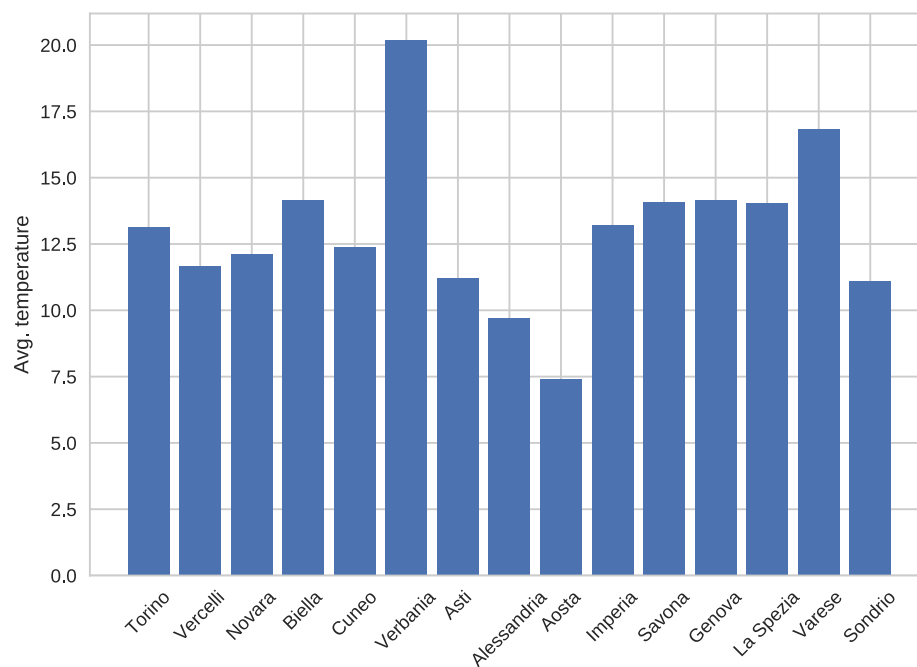

In line 2 we create a bar plot using the `bar()` function passing **province** to specify the labels for the x axis and **T** to specify the values for the y axis. We have not specified a position for the columns, so Matplotlib will arrange them automatically. In line 3 we use `draw()` to draw the plot. `draw()` doesn't actually display the graph (for that there is `show()`), but it initializes all the elements inside the plot. We need to do this because in line 4 we use `ax.get_xticklabels()`, which returns a list of strings containing the labels of the ticks for the x axis. If we had not called `plt.draw()` in the line before, we would have obtained an empty list. By passing said list to `ax.set_xticklabels()` and passing 45 to **rotation** we are able to rotate the labels by 45Â°.

> **NOTE**
>
> Instead of calling `plt.draw()` and then using `ax.get_xticklabels()`, we could have simply passed **province** to `ax.set_xticklabels()`.

To plot more than one array of data and group the columns by province we have to tell matplotlib the position and width of each column:

```
 1  barWidth=0.3
 2  r1=np.arange(len(province[:15]))
 3  r2=r1+barWidth
 4
 5  fig, ax = plt.subplots()
 6  ax.bar(r1, daily[:15], width=barWidth, label="daily avg.")
 7  ax.bar(r2, ndays[:15], width=barWidth, label="days > 10mm")
 8  ax.set_xticks(r1)
 9  ax.set_xticklabels(province[:15], rotation=45)
10  ax.set_ylabel("Avg. temperature")
11  ax.legend()
12  plt.show()
```
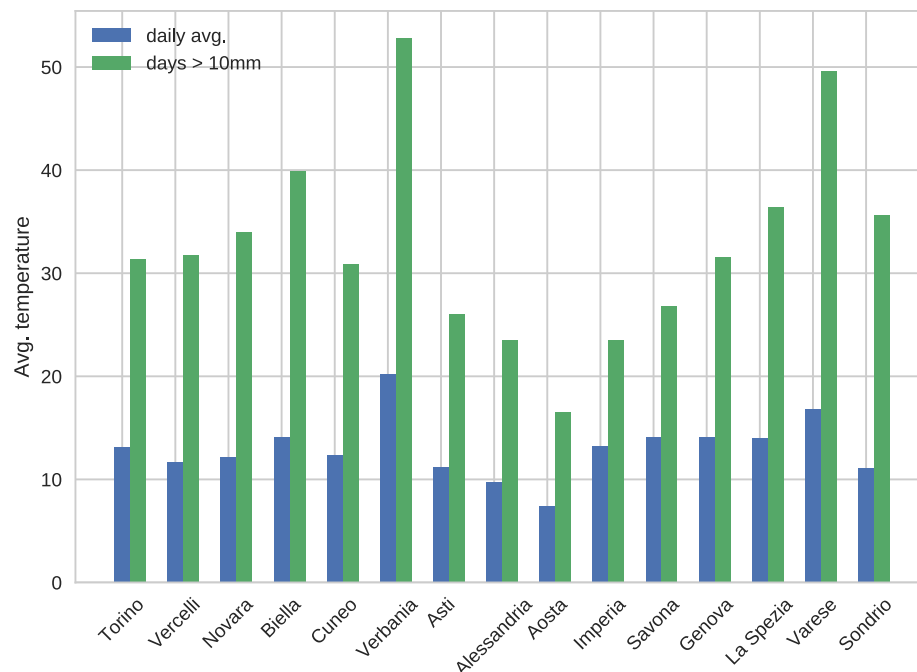

The variable **barWidth** stores the width of the bars. in line 2 and 3 we create two arrays that store the position of each bar that we wish to plot. Notice how the values of **r2** have an offset from the values in **r1** equal to **barWidth**. In lines 5 and 6 we create the bar plots, specifying a custom position for the bars by passing **r1** and **r2** as first arguments. The **width** parameter is used to control the width of the bars, so we pass **barWidth** to it. The next thing to do is to set up the x axis so that it gets displayed properly. Using `set_xticks()` we define the position of the ticks and with `set_xticklabels()` we tell matplotib to use the strings stored in **province** as labels and to rotate them by 45Â°.

APPENDIX: parameter references

Named colors

black	k	dimgray	dimgrey
gray	grey	darkgray	darkgrey
silver	lightgray	lightgrey	gainsboro
whitesmoke	w	white	snow
rosybrown	lightcoral	indianred	brown
firebrick	maroon	darkred	r
red	mistyrose	salmon	tomato
darksalmon	coral	orangered	lightsalmon
sienna	seashell	chocolate	saddlebrown
sandybrown	peachpuff	peru	linen
bisque	darkorange	burlywood	antiquewhite
tan	navajowhite	blanchedalmond	papayawhip
moccasin	orange	wheat	oldlace
floralwhite	darkgoldenrod	goldenrod	cornsilk
gold	lemonchiffon	khaki	palegoldenrod
darkkhaki	ivory	beige	lightyellow
lightgoldenrodyellow	olive	y	yellow
olivedrab	yellowgreen	darkolivegreen	greenyellow
chartreuse	lawngreen	honeydew	darkseagreen
palegreen	lightgreen	forestgreen	limegreen
darkgreen	g	green	lime
seagreen	mediumseagreen	springgreen	mintcream
mediumspringgreen	mediumaquamarine	aquamarine	turquoise
lightseagreen	mediumturquoise	azure	lightcyan
paleturquoise	darkslategray	darkslategrey	teal
darkcyan	c	aqua	cyan
darkturquoise	cadetblue	powderblue	lightblue
deepskyblue	skyblue	lightskyblue	steelblue
aliceblue	dodgerblue	lightslategray	lightslategrey
slategray	slategrey	lightsteelblue	cornflowerblue
royalblue	ghostwhite	lavender	midnightblue
navy	darkblue	mediumblue	b
blue	slateblue	darkslateblue	mediumslateblue
mediumpurple	rebeccapurple	blueviolet	indigo
darkorchid	darkviolet	mediumorchid	thistle
plum	violet	purple	darkmagenta
m	fuchsia	magenta	orchid
mediumvioletred	deeppink	hotpink	lavenderblush
palevioletred	crimson	pink	lightpink

Linetypes

`':'`

`'-.'`

`'--'`

`'-'`

Markers
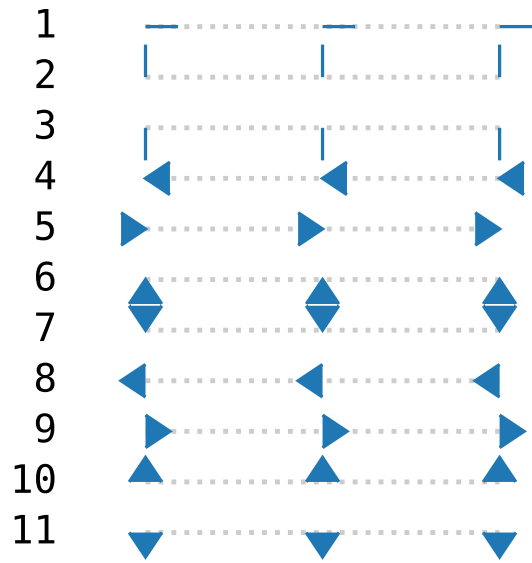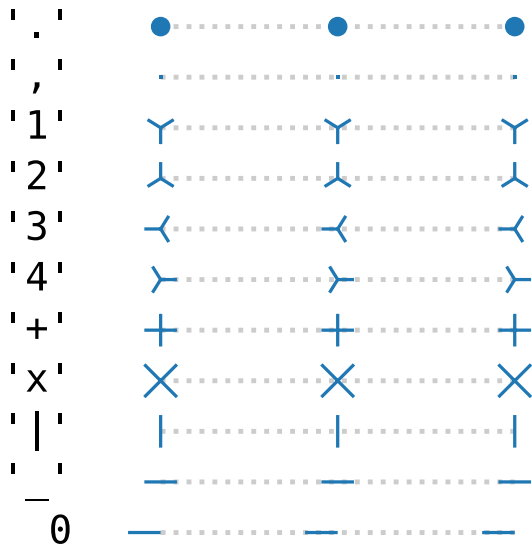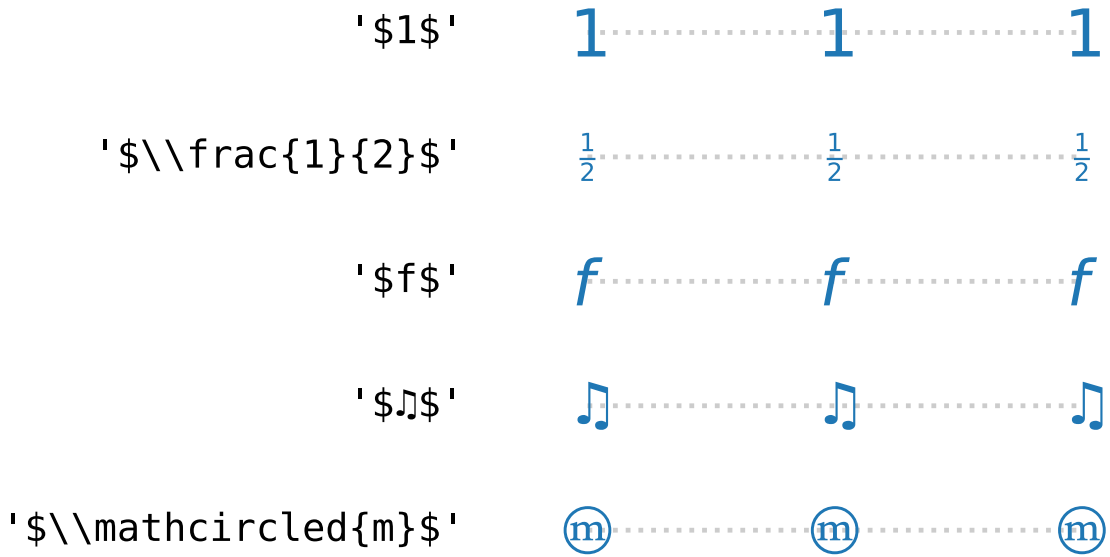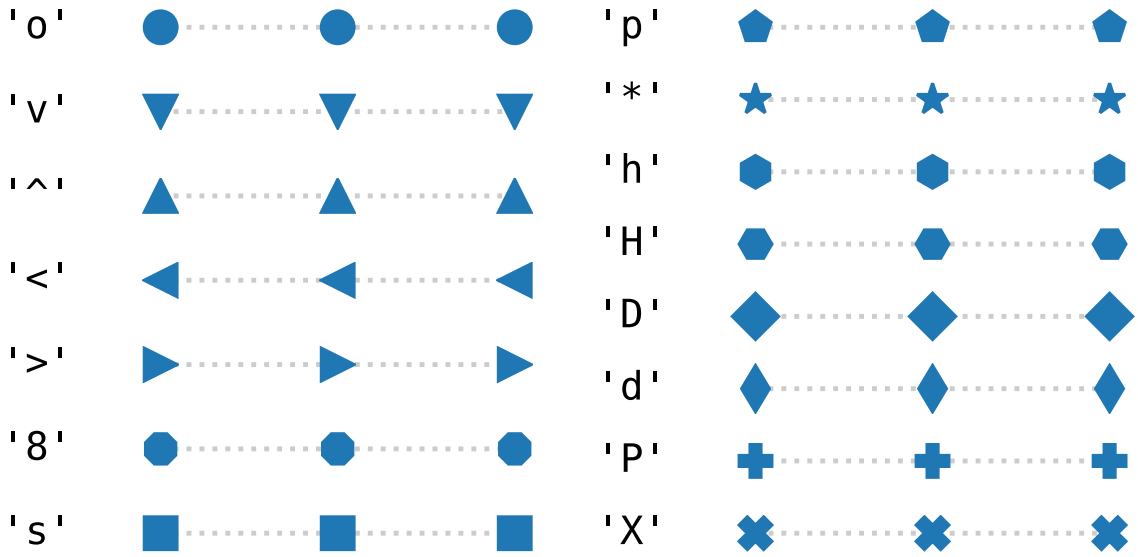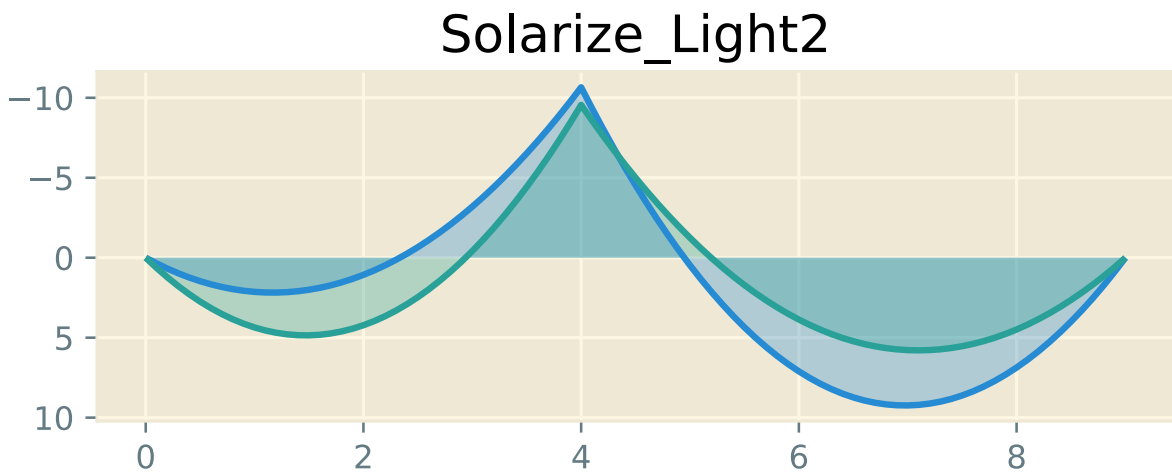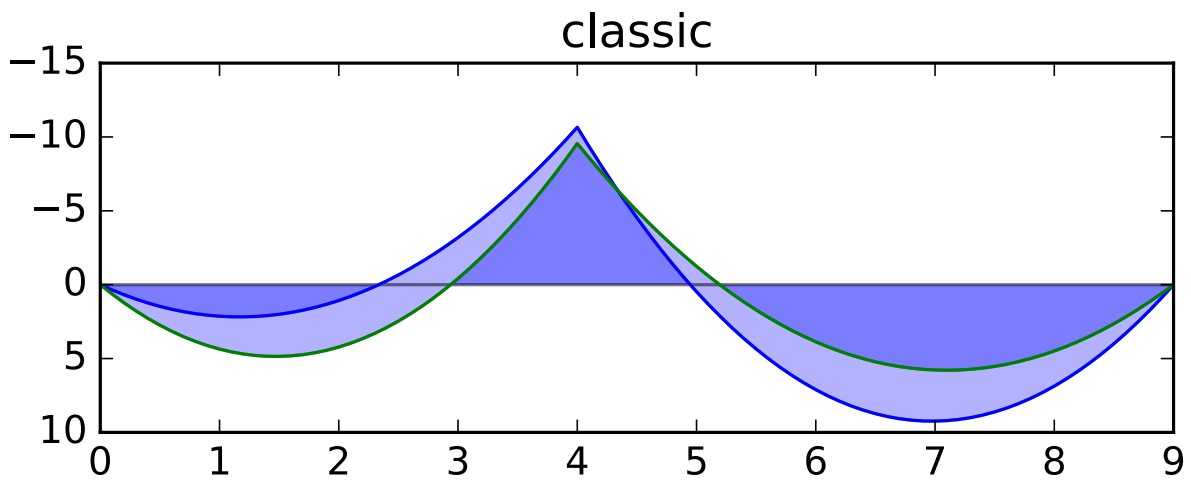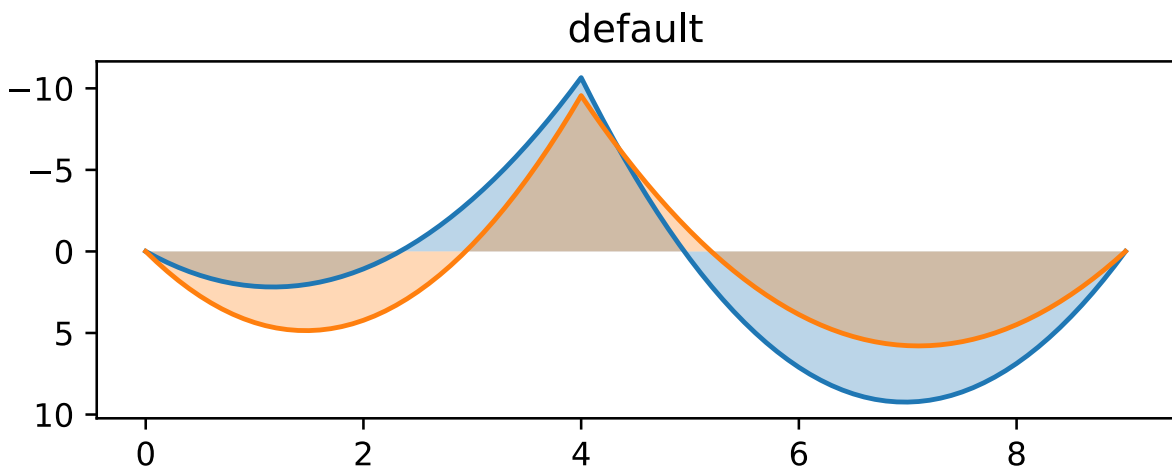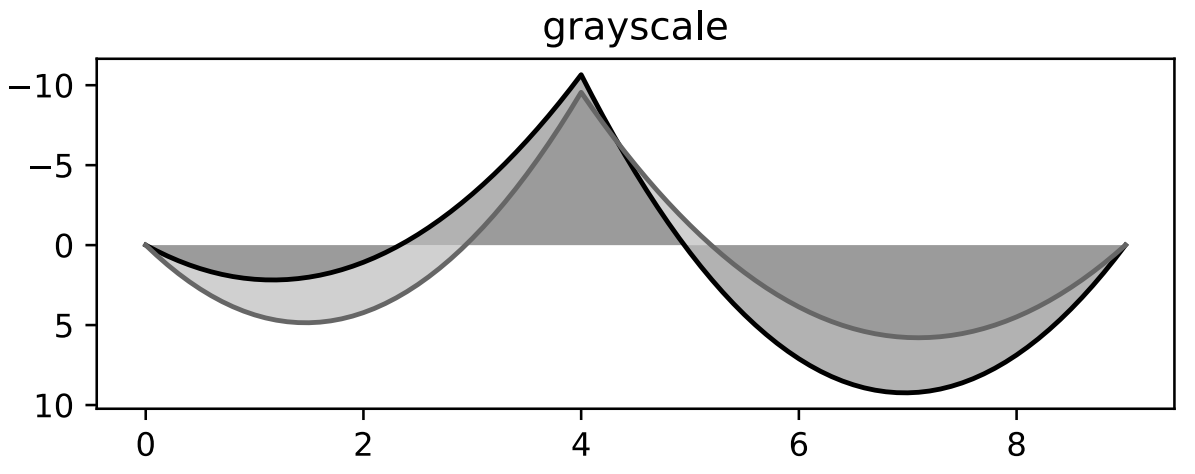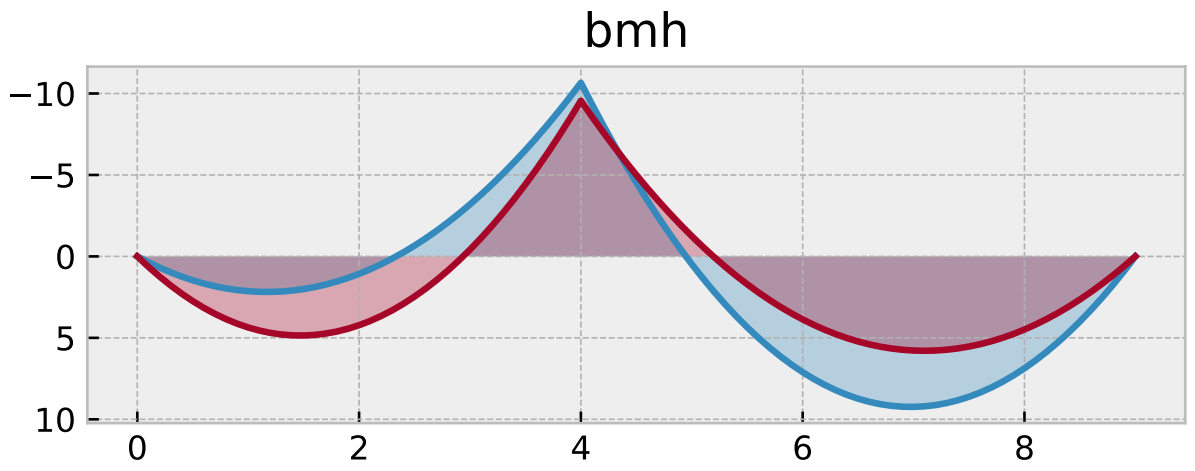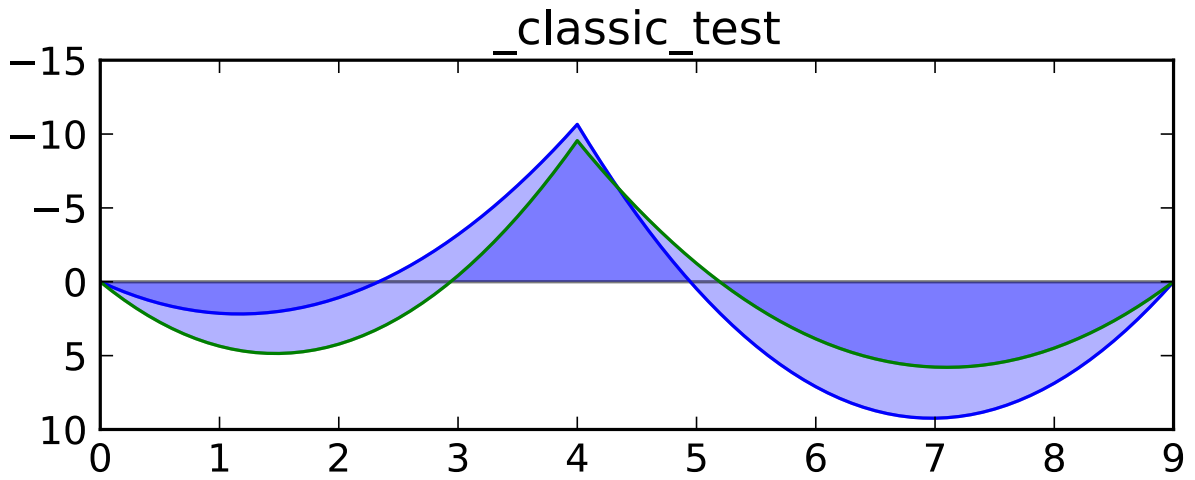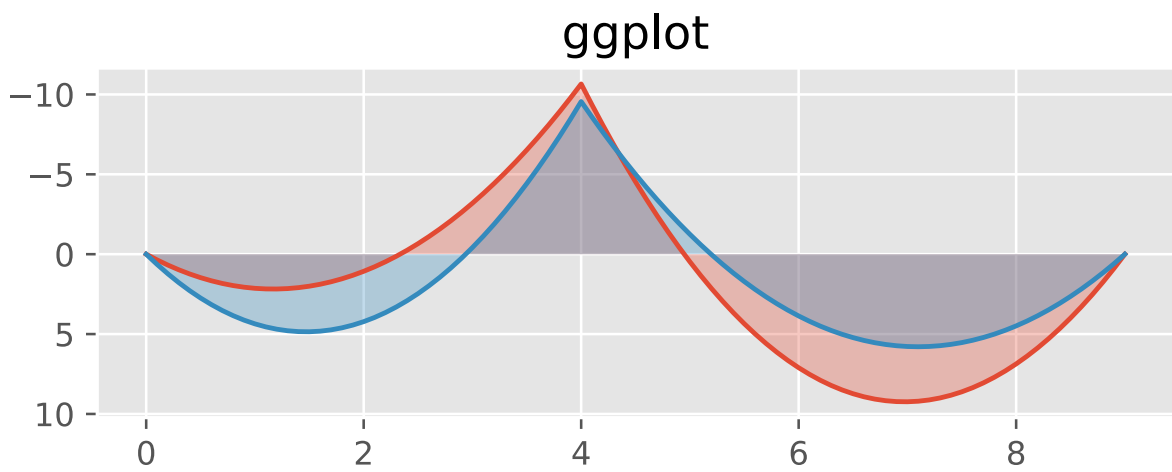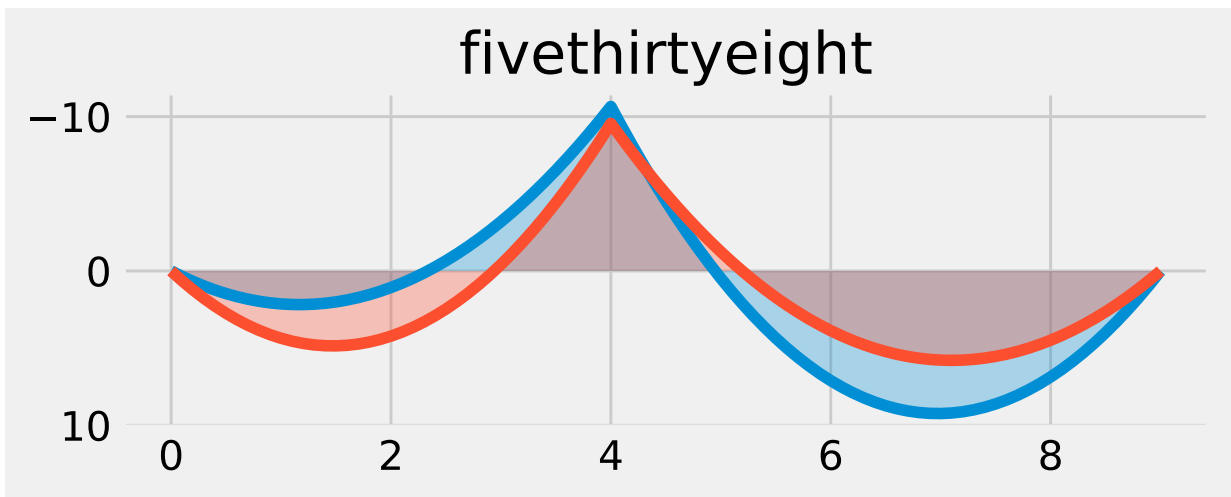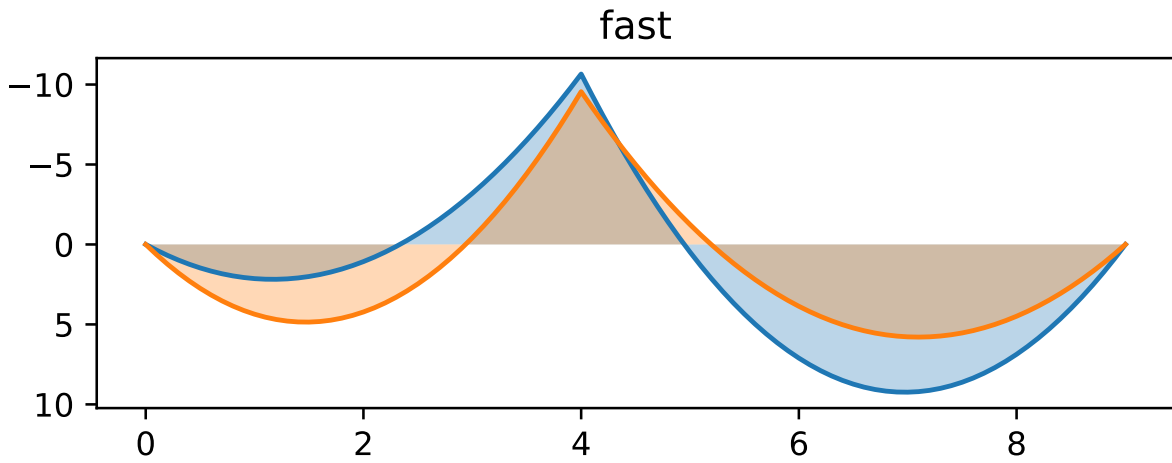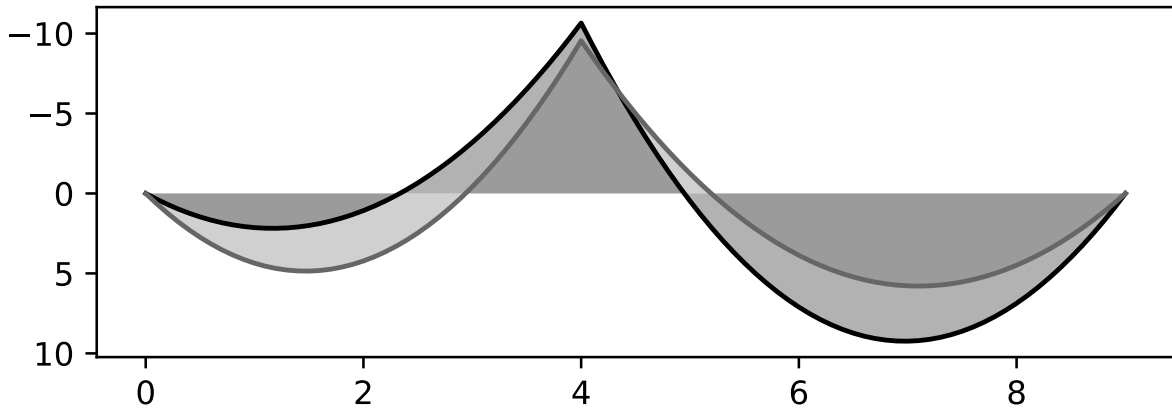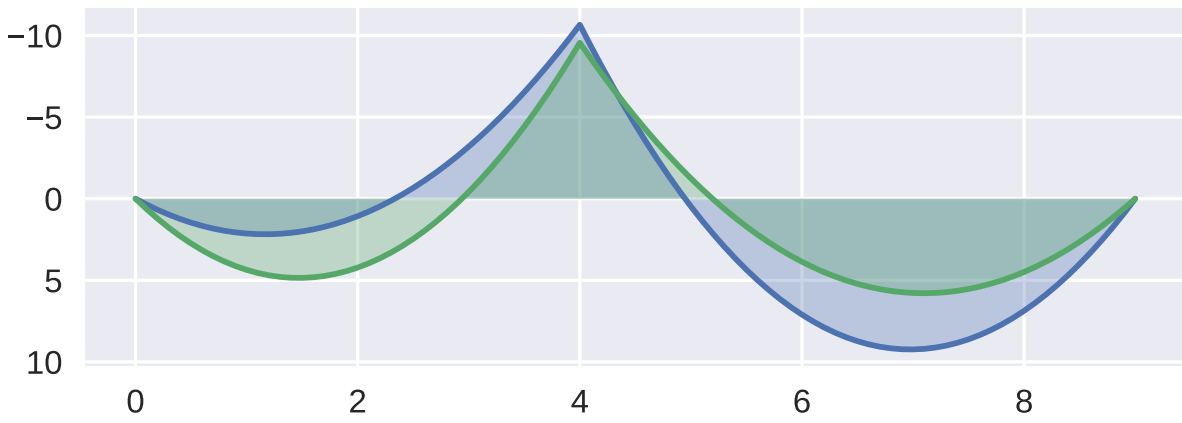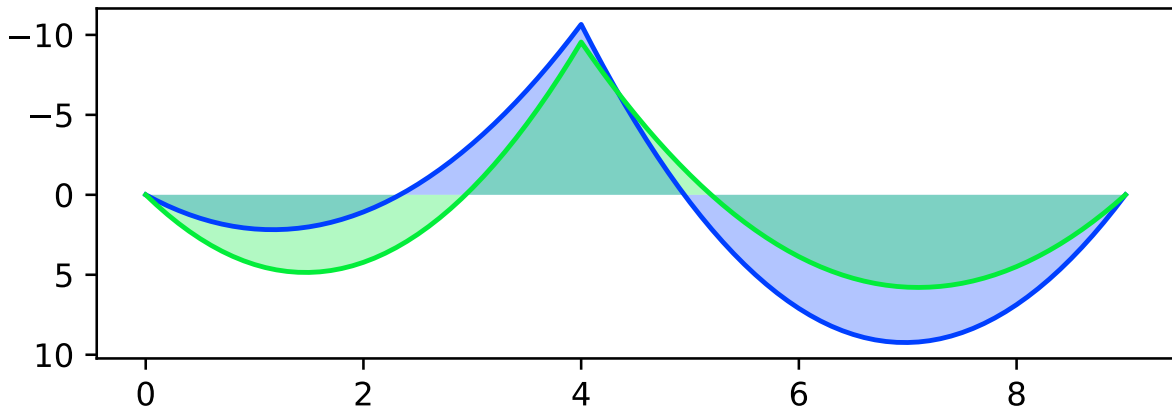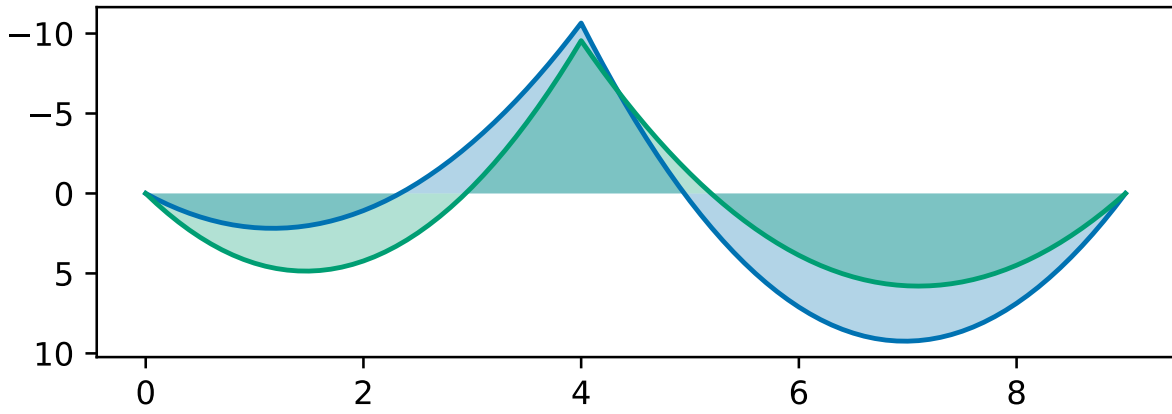
'o' ● ● ●   'p' ⬟ ⬟ ⬟
'v' ▼ ▼ ▼   '*' ★ ★ ★
'^' ▲ ▲ ▲   'h' ⬢ ⬢ ⬢
'<' ◀ ◀ ◀   'H' ⬣ ⬣ ⬣
'>' ▶ ▶ ▶   'D' ◆ ◆ ◆
'8' ⯃ ⯃ ⯃   'd' ◆ ◆ ◆
's' ■ ■ ■   'P' ✚ ✚ ✚
            'X' ✖ ✖ ✖

'1' 1 ⋯ 1 ⋯ 1

'$\\frac{1}{2}$' $\frac{1}{2}$ ⋯ $\frac{1}{2}$ ⋯ $\frac{1}{2}$

'f' f ⋯ f ⋯ f

'$♫$' ♫ ⋯ ♫ ⋯ ♫

'$\\mathcircled{m}$' ⓜ ⋯ ⓜ ⋯ ⓜ

Plot Styles

default

classic

Solarize_Light2

_classic_test

bmh

grayscale
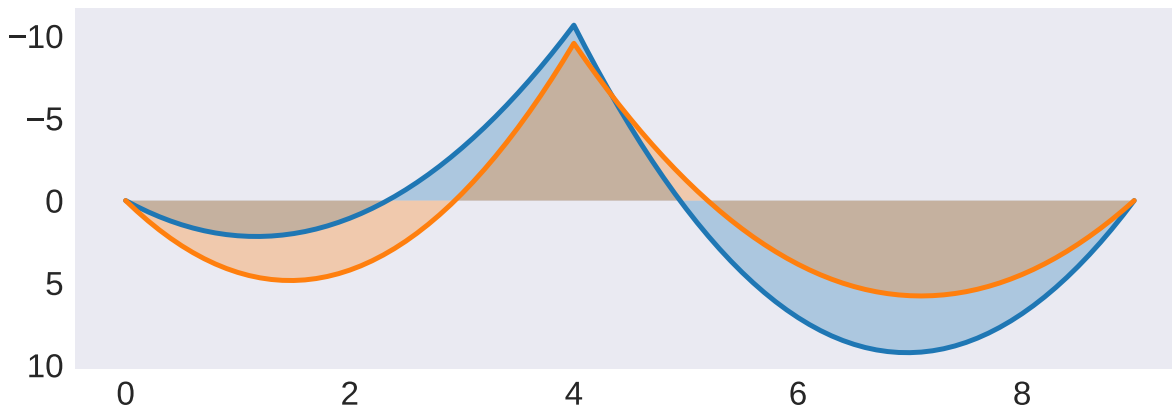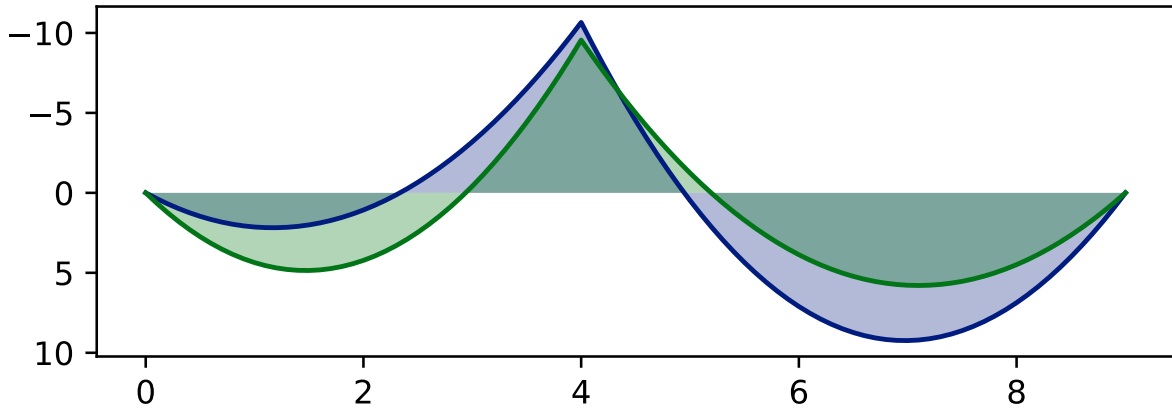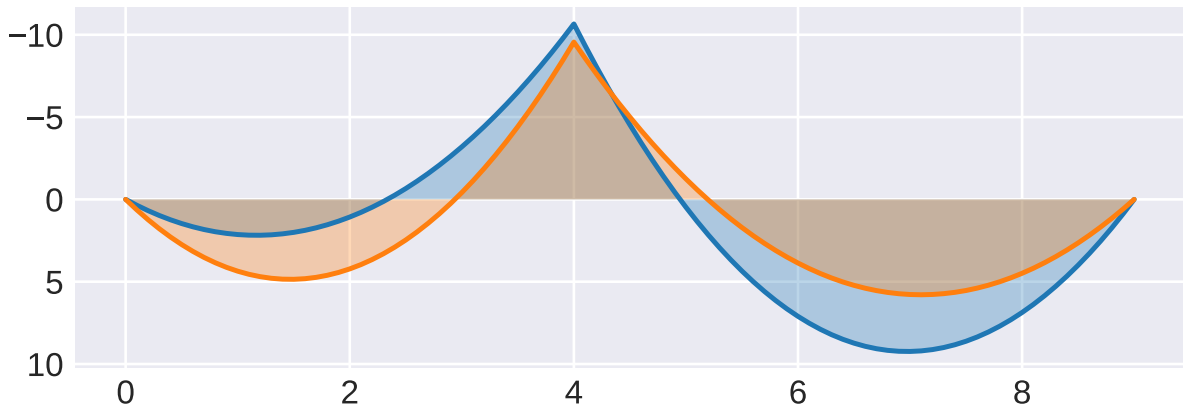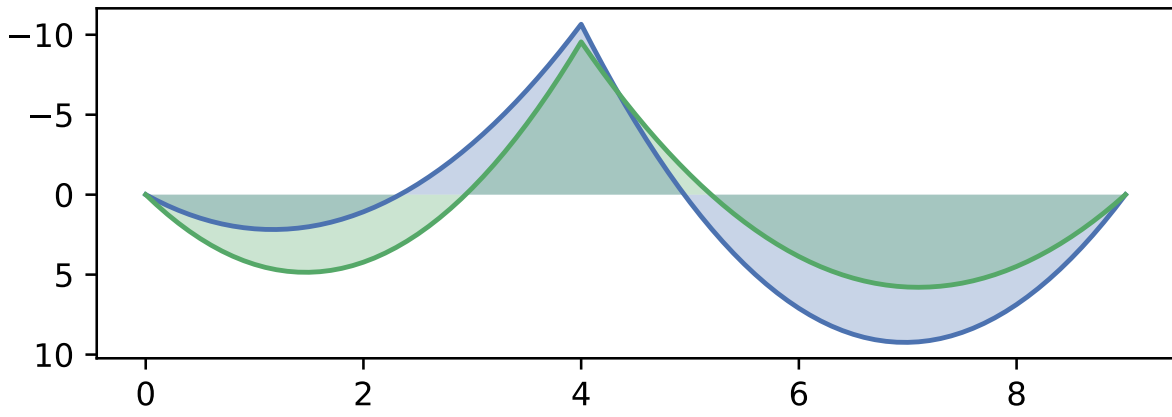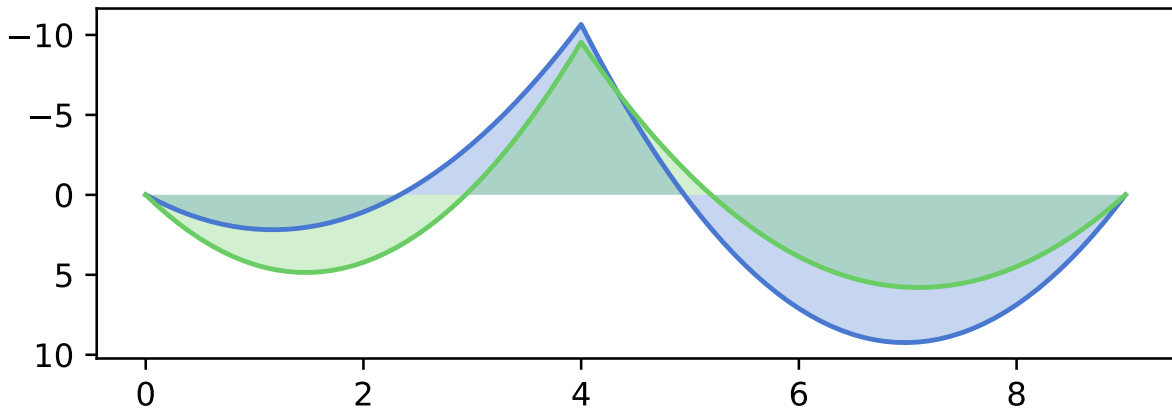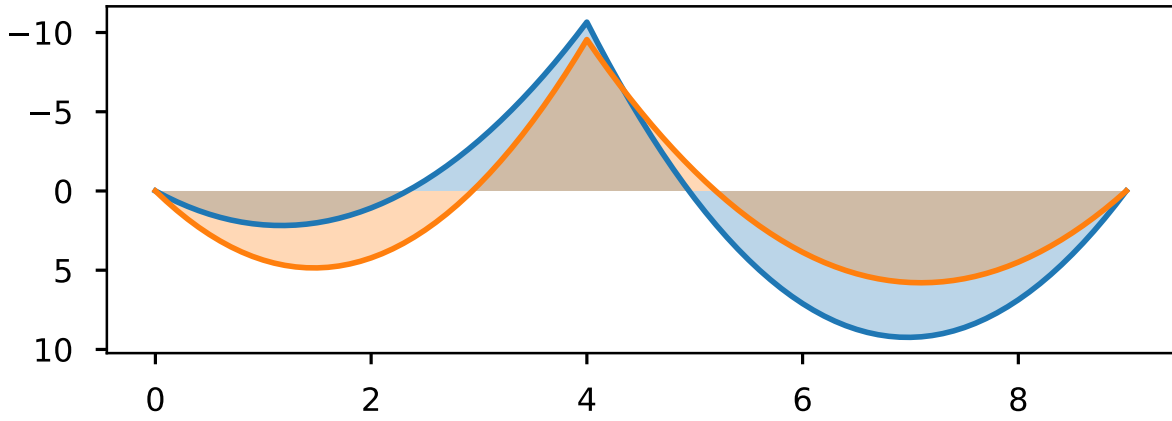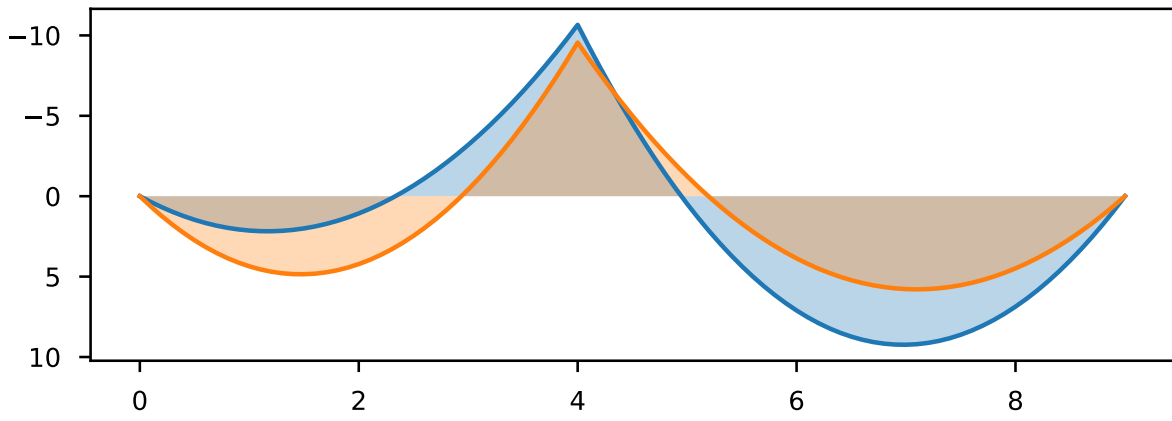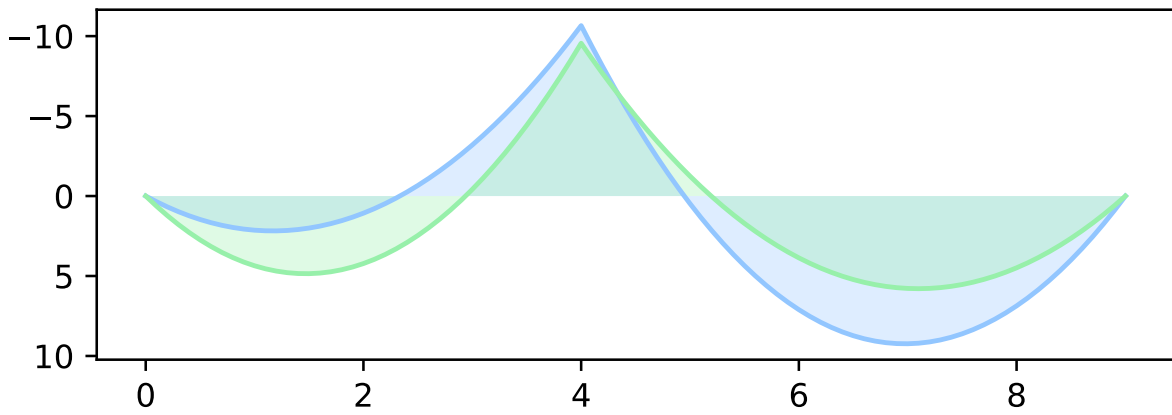
seaborn-poster
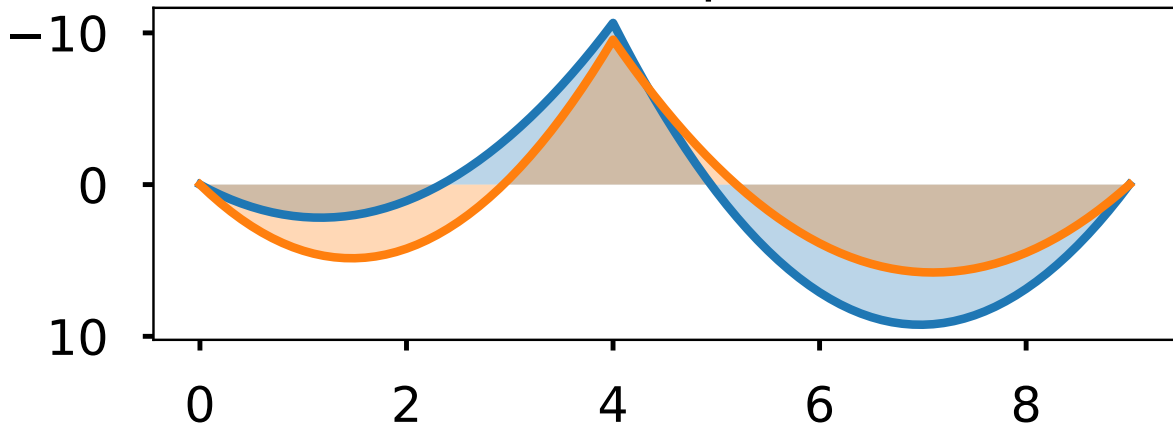
seaborn-talk
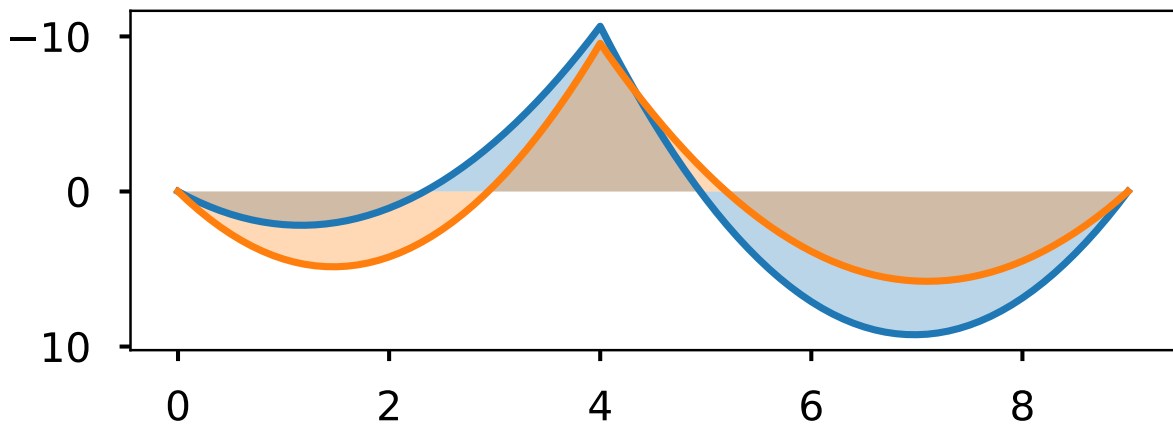
seaborn-ticks
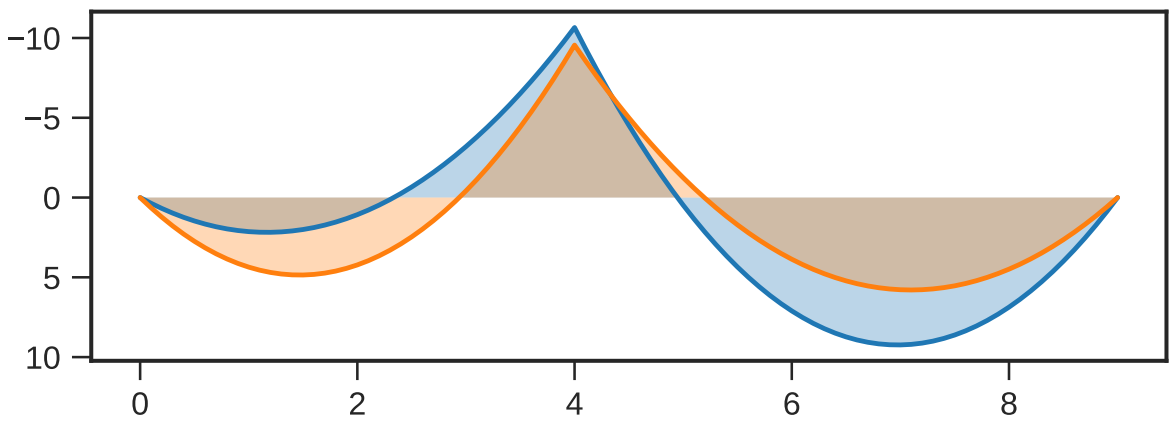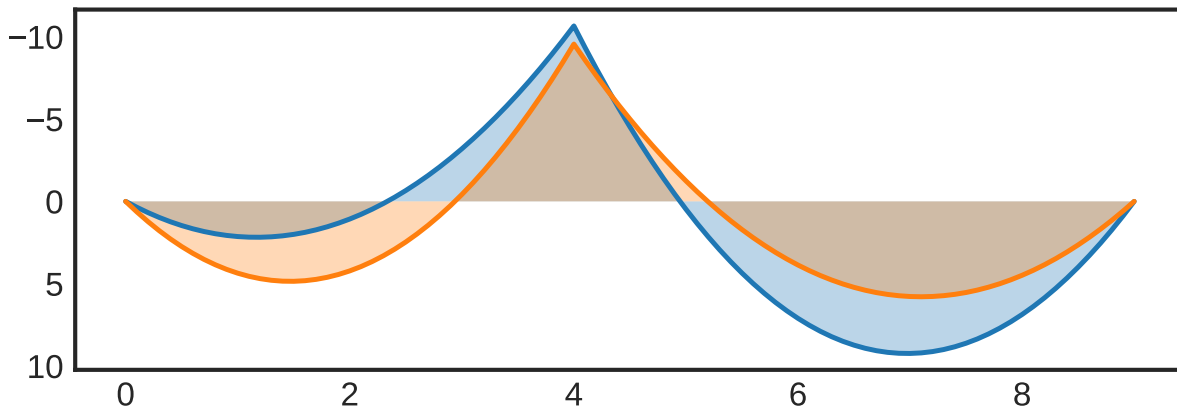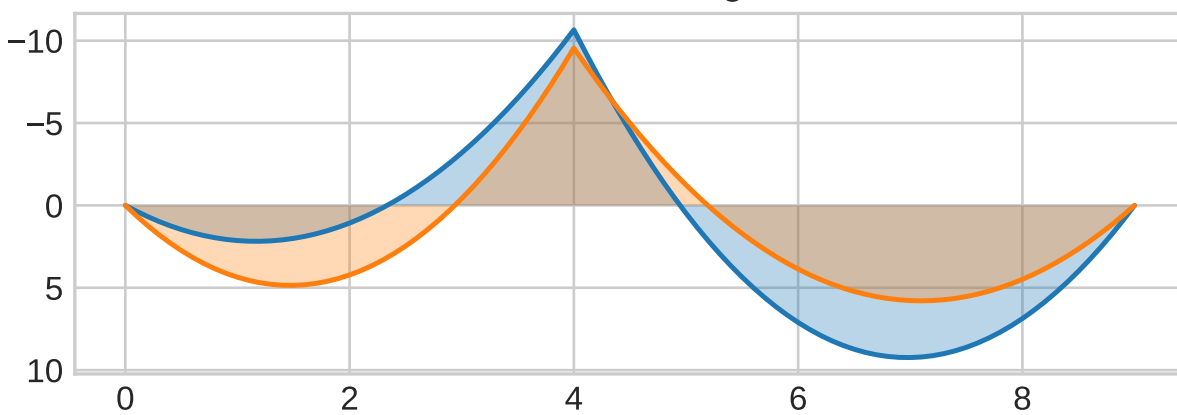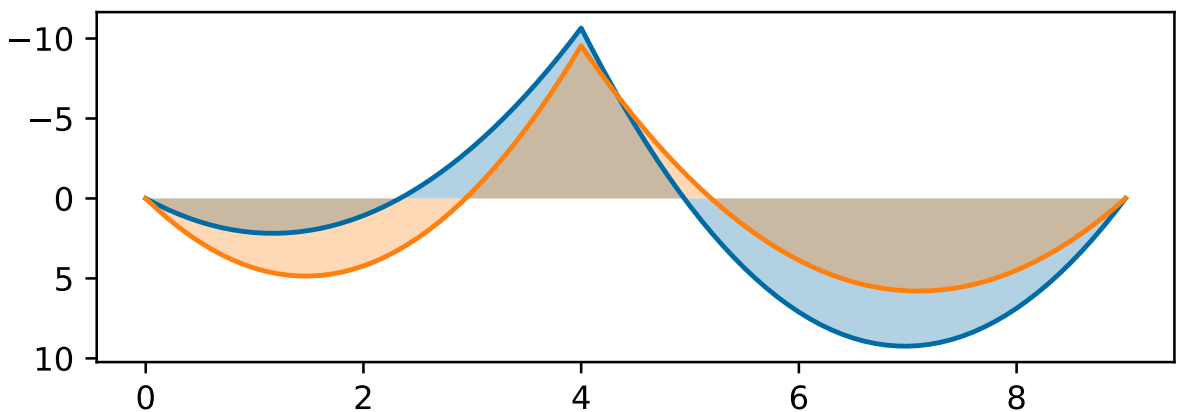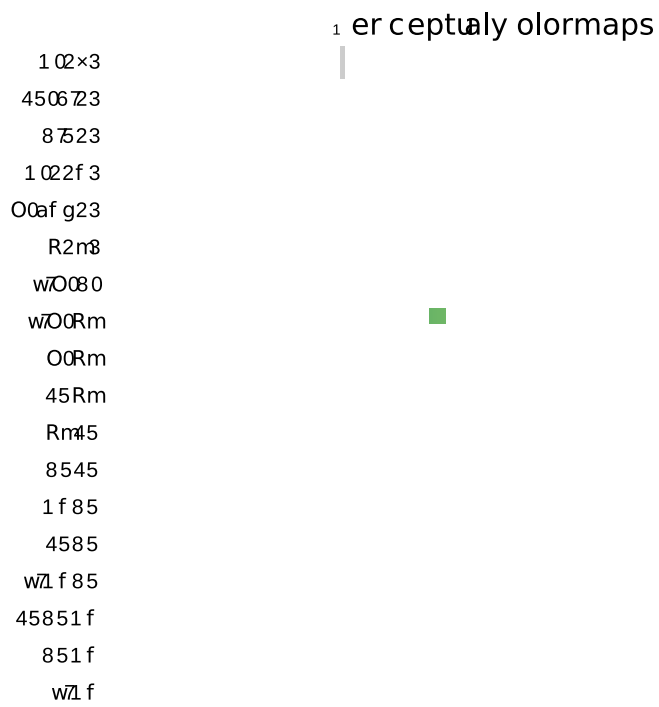
Color maps

erceptually or nps rm pesuer tnal cslsrmap

10 20×03	
45 637 6	
08 fO28 a	
7 6g7 6	
R010×08	

er ceptualy olormaps

1 02×3	
4506723	
8 7523	
1 022f 3	
O0af g23	
R2m8	
w7O08 0	
w7O0Rm	
O0Rm	
45Rm	
Rm45	
8545	
1 f 85	
4585	
w71 f 85	
45851 f	
851 f	
w71 f	

er ceptualy orynplpsmaps

er cpteutal y ypo r mp

er portrucual y o

₁ er qxtr tpualy c or r mp

102×3 45
102×3 46
10 7 83 f
Oaa3 g×
R 08 m 6
w3 ×5
w3 ×6
w3 ×t
×0a5 o
×0a6 o
×0a6 oa
×0a6 oa

₁ er cpttuapl yr cl tl or umr

102×
3 45 6 7
8 f O2 a
×56 gRO2 4 gm
gO4 42 5 a
×56 gR6 gO4 a
×aw3 08 g
×aw3 08 gt
a o w7 23
f wm Om O 05 x
m4 ×
×56 gR42 5 am8 w
42 5 am8 w
Og
a53 R63 Of g420
×56 gRa f 24
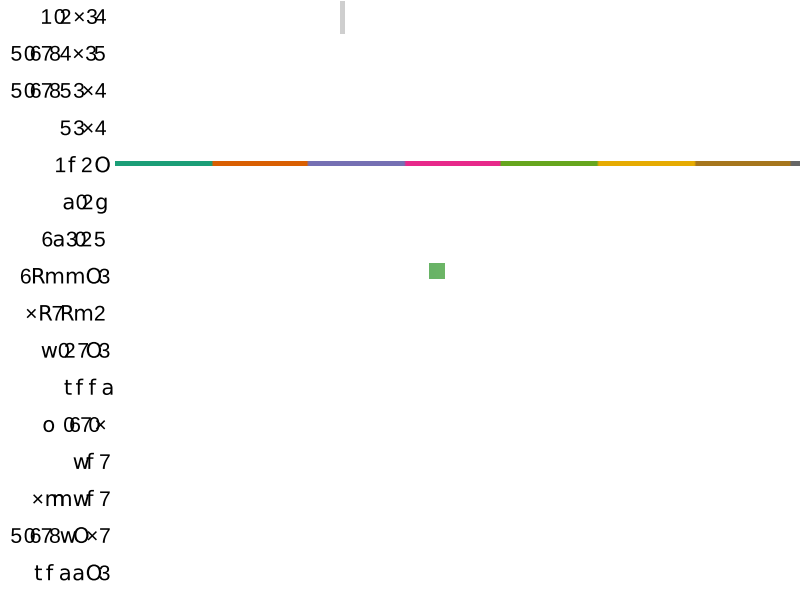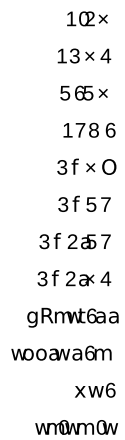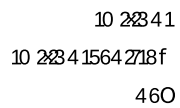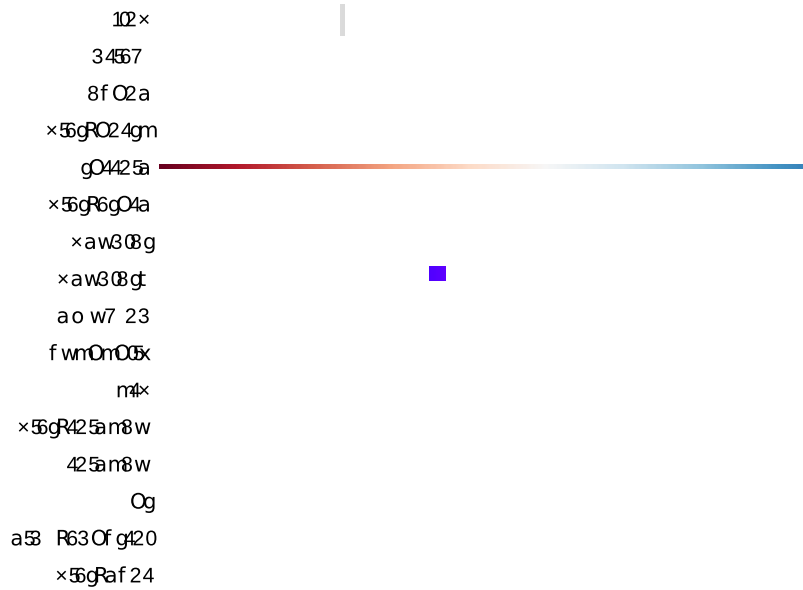
Calculating section properties

We finally have all the tools necessary to start solving real-world problems . The libraries introduced previously will play a vital role in every example of this second part of the book, so please refer to the first if you forget some of the commands that you might encounter.

Section properties of a steel beam

In this example we will consider a welded I beam with asymmetric flanges. The properties that will be calculated are:

- Area A
- Section centroid y_G
- first moment of area about the x axis S_x
- first moment of area about the y axis S_y
- second moment of area about the x axis I_x
- second moment of area about the y axis I_y
- elastic section modulus about the x axis W_x
- elastic section modulus about the y axis W_y
- elastic moment of resistance about the x axis $M_{Rd,x}$
-

The quantities that define the geometry of the section are displayed in the next figure:
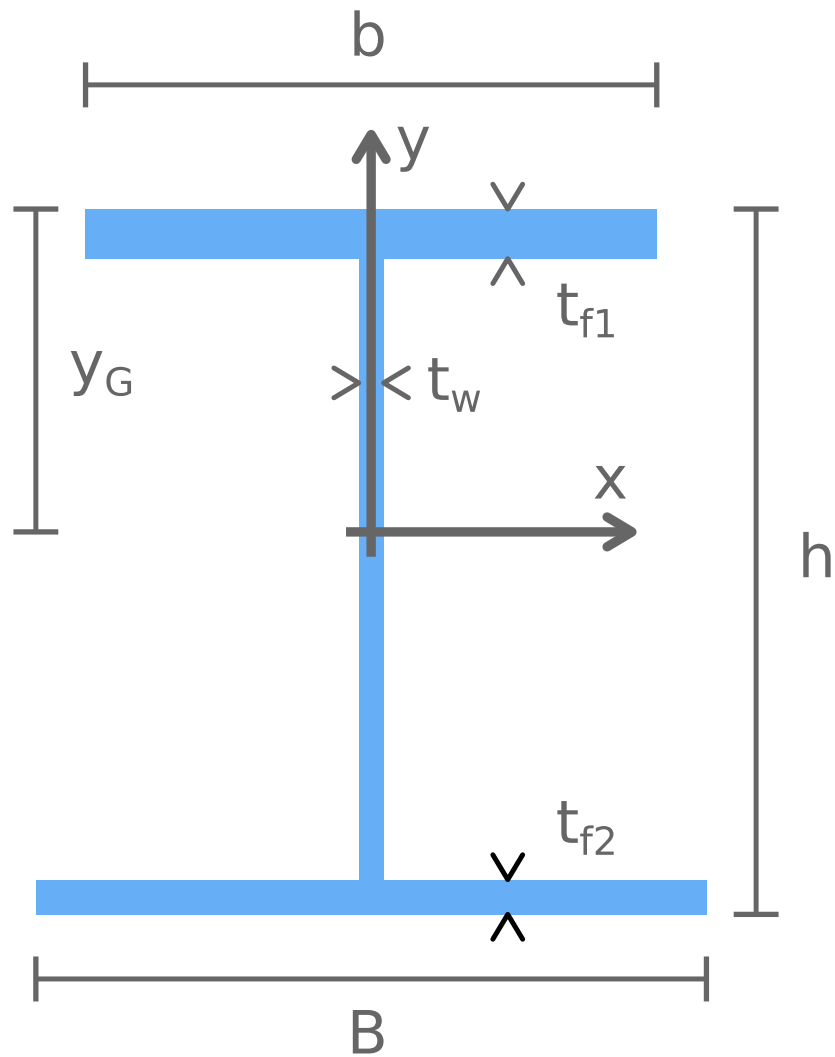
In this example we will consider the following values:

Name	Value	Unit
b	250	mm
B	300	mm
h	400	mm
t_{f1}	18	mm
t_{f2}	15	mm

t_w	12	mm

Let's open up a new jupyter notebook. In the first cell we will import all the libraries that we will need:

```python
1  import numpy as np
2  import sympy as sp
3  import pandas as pd
4
5  sp.init_printing(use_latex="mathjax")
```

In order to display the results in latex format, we call **init_printing** from SymPy. Once the libraris are loaded, the first thing to do is to store the dimensions of the section into variables:

```python
1  b=250  #mm
2  B=300  #mm
3  h=400  #mm
4  tf1=18  #mm
5  tf2=15  #mm
6  tw=12  #mm
7
8  hw=h-tf1-tf2
```

It is good practice to write the unit of measurement as a comment next to every variable. This might seem superfluous in a simple example like this, however when the notebook starts to become more crowded it will be essential in order to avoid mistakes. The section is divided in three parts: the top flange, the bottom flange and the web. We now store the area of each part in three different variables, and calculate the total area of the section:

```python
1  Af1=b*tf1
2  Af2=B*tf2
3  Aw=hw*tw
4  A=Af1+Af2+Aw
5  A
```

Next we calculate y_G, which is the distance of the centroid from the top of the beam. We know that the first moment of area of the portion of the beam above y_G must be equal to the one of the portion below, meaning that

$$b \cdot t_{f1} \cdot \left(y_G - \frac{t_{f1}}{2}\right) - B \cdot t_{f2} \cdot \left(h - y_G - \frac{t_{f2}}{2}\right) + \frac{t_w \cdot (y_G - t_{f1})^2}{2} - \frac{t_w \cdot (h - y_G - t_{f2})^2}{2} = 0$$

> **NOTE**
>
> The first moment of area for a rectangle is given by the area of the rectangle multiplied by the distance of its centroid from the axis taken in consideration. In this case we must calculate the first moment of area relative to y_G, which at the moment is still an unknown quantity.

The equation above must be solved with respect to y_G, so in order to do this we will have to use SymPy. It is important to note that we are assuming the position of the centroid to be somewhere inside the web of the beam. If this wasn't the case, we would need to change the equations accordingly.

```
1  yG=sp.symbols('yG')
2  Stot=b*tf1*(yG-tf1/2)-B*tf2*(h-yG-tf2/2)\
3               +tw*(yG-tf1)**2/2-tw*(h-yG-tf2)**2/2
4  yG=sp.solveset(Stot, yG).args[0]
5  yG
```

```
200.996418979409
```

In line 1 we define a SymPy symbol called **yG**, and in line 2 we write the formula of the first moment of area. Every quantity that appears in this formula is known, except **yG**. The result is a SymPy expression that we store in **Stot**. In line 4 using `sp.solveset()` we solve the equation. Remember that SymPy automatically adds =0 when using `solveset`. Using `args[0]` we access the first (and only) element stored inside the output of `solveset`, and we assign it to **yG**.

Now we can use **yG** to calculate the first moment of area about the x axis, by considering only the top half of the section.

```
1  Sx=b*tf1*(yG-tf1/2)+tw*(yG-tf1)*(yG-tf1)/2
2  Sx
```

The section is symmetric in the horizontal direction, therefore the centroid **yG** is located in the middle of the web. We can calculate the first moment of area about the y axis by considering the left or right half of the section:

```
1  Sy=tf1*(b/2)*(b/4)+tf2*(B/2)*(B/4)+hw*(tw/2)*(tw/4)
2  Sy
```

```
315981.0
```

Next, we calculate the second moment of area about the x and y axes. Like before we consider the section as a collection of rectangles. Each rectangle contributes to the total second moment of area of the section according to the following formula:

$$I_i = \frac{b_i \cdot h_i^3}{12} + A_i \cdot d_i^2$$

Where b_i, h_i and A_i are the width, height and area of the rectangle, and d_i is the distance between the centroid of the rectangle and the axis we are considering. To implement this formula in the code we calculate I separately for the two flanges and the web, and then we sum the results.

```
1  Ix_f1=(b*tf1**3)/12+Af1*(yG-tf1/2)**2
2  Ix_f2=(B*tf2**3)/12+Af2*(h-yG-tf2/2)**2
3  Ix_w=(tw*hw**3)/12+Aw*(tf1+hw/2-yG)**2
4  Ix=Ix_f1+Ix_f2+Ix_w
5  Ix
```

```
380550963.828111
```

The same thing can be done to calculate the second moment of area about the y axis:

```
1  Iy_f1=(tf1*b**3)/12
2  Iy_f2=(tf2*B**3)/12
3  Iy_w=(hw*tw**3)/12
4  Iy=Iy_f1+Iy_f2+Iy_w
5  Iy
```

57240348.0

Now we can calculate the **elastic modulus**, defined as

$$W_e = \frac{I}{z}$$

where I is the second moment of area and z is the distance between the outer-most fibre of the section and the centroid. So if we consider the x axis:

```
1  Wx=min(Ix/(yG), Ix/(h-yG))
2  Wx
```

1893322.10872422

Notice how since we don't know if the outer-most fibre is in the top or the bottom flange, so we write the formulas for both scenarios and then store into **Wx** only the lowest value.

The last quantity that we are going to calculate is the moment of resistance, calculated as

$$M_x = W_x \cdot f_y$$

where f_y is the yeld strength of the steel. For this easy introductionary example we will omit all the safety coefficients and use the bare formula found in solid mechanics theory. In the following chapters you will find more in-depth examples. Considering **S235** as the steel grade we can apply the formula above:

```
1  fy=235  #N/mm^2
2  Mx=Wx*fy*10E-3  #kNm
3  Mx
```

4449306.95550191

The next table summarizes all the quantities found so far:

Name	Value	Unit
A	13404	mm^2
yG	200.99	mm
Sx	1064910.02	mm^3
Sy	315981	mm^3
Ix	380550963.82	mm^4
Iy	57240348	mm^4
Wx	1893322.10	mm^3
Mx	4449306.95	kNm

Conclusions

If you have followed the steps above you now have a functioning jupyter notebook capable of calculating the cross-section properties of a welded steel beam. All we had to do was to implement some formulas, so the code was not that complex. Nevertheless, it still gives a comprehensive insight on how to use python to translate theory into practice. In the next chapter you will tackle a much more difficult problem, designed to make you understand how to use **functions** in order to organize the different parts of your code.

M-N interaction in a concrete section

Ultimate Limit States (ULS)

In this chapter we will calculate the bending-axial force limit domain for a rectangular concrete beam. The methodology will follow the Ultimate Limit States (ULS) design filosofy, so the goal is to obtain the bending moment resistance and the axial load resistence of any given section. The ULS criterion is summarized with the following expression:

$$\frac{R_d}{S_d} \geq 1.0$$

where S_d represents a general design load and R_d the resistance of the structure. The final result will be a graph where the horizontal axis is the axial force, and the vertical axis is the moment of resistance of the cross-section. It looks something like this:
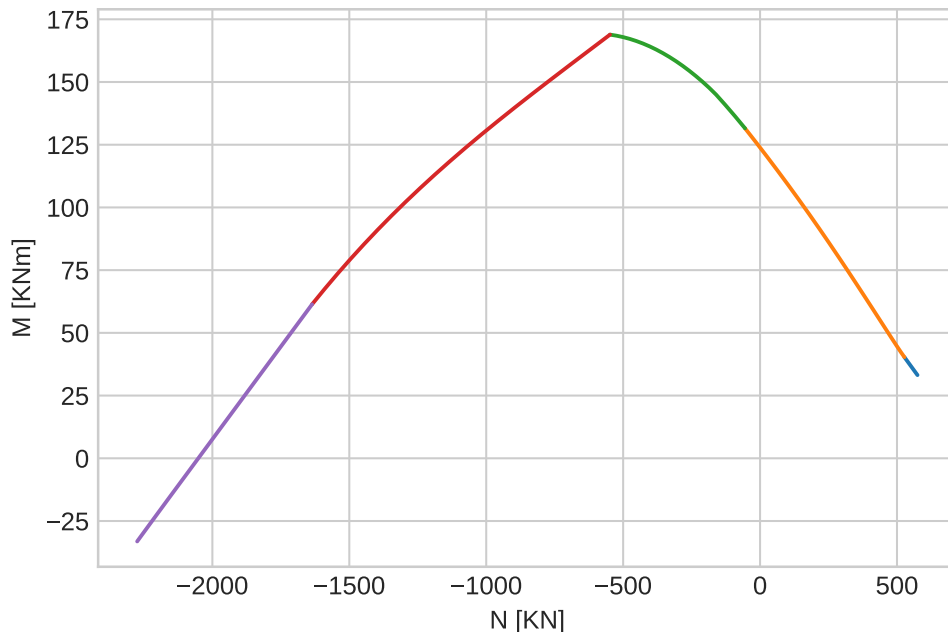
We will make the following assumptions:

- The sections remain planar as the beam deflects. This is equal to ignoring the deflection due to the shear force.
- The rebars remain perfectly adherent to the concrete, meaning that steel has the same strain as the concrete.
- The contribute of the concrete under tension to the overall resistance of the section will be ignored.
- Tensile forces and strains are considered to be *positive*
- Compressive forces and strains are considered to be *negative*
- Bending moments are considered to be *positive* when they stretch the bottom half of the section

Materials and geometry

Throughout this chapter we will consider a concrete section with the following dimensions:
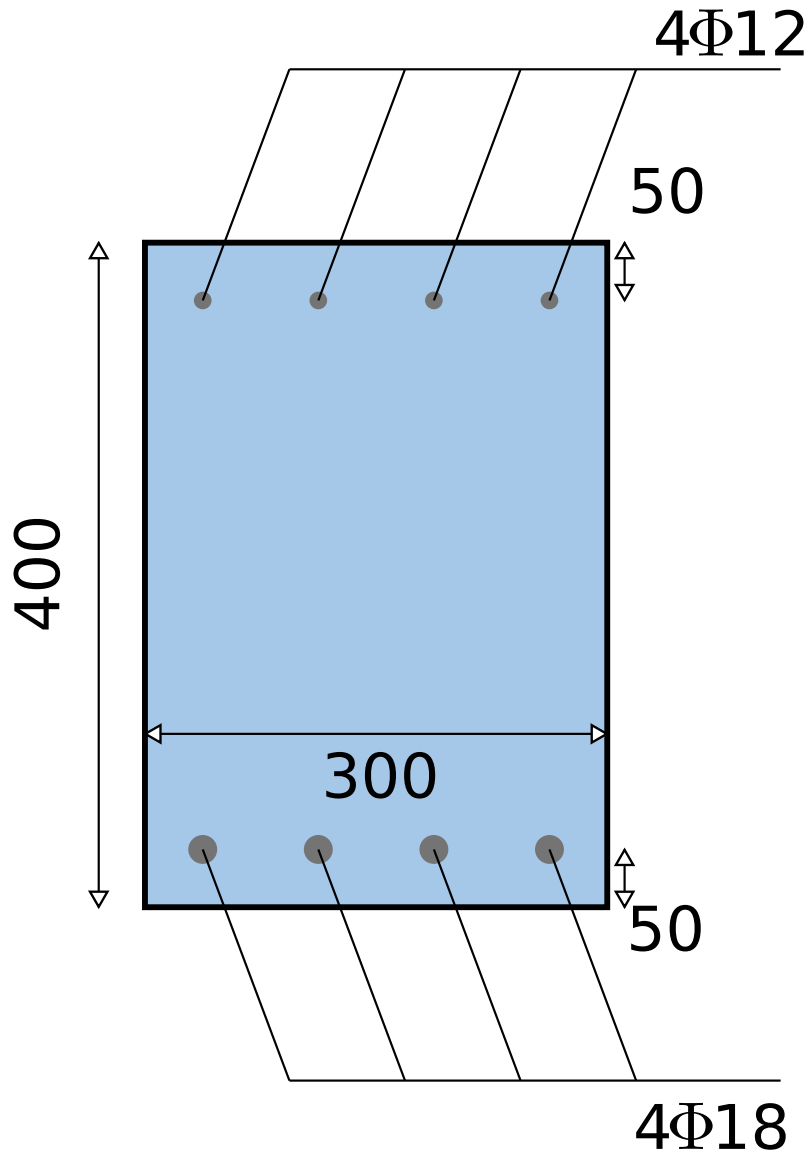
The measures are in mm. The concrete is a standard C25/30 (f_{ck}=25 MPa and $f_{ck,cube}$=30 MPa) and the rebar steel is a standard B450C with yeld strength f_{yk}=450 MPa. Next we define the full stress-strain relationship for both materials.

Concrete

The design stress-strain relationship for the concrete has an initial parabolic increment and a subsequent horizontal branch once the yeld stress is reached, as shown in the next figure:

$$\varepsilon_{cu} = -3.5E-3 \qquad \varepsilon_{cc} = -2.0E-3$$

$$\sigma_c = -f_{cd}\left(2 - \frac{\varepsilon_c}{\varepsilon_{cc}}\right)\frac{\varepsilon_c}{\varepsilon_{cc}}$$

$$-f_{cd} = \frac{-0.85 f_{ck}}{\gamma_c}$$
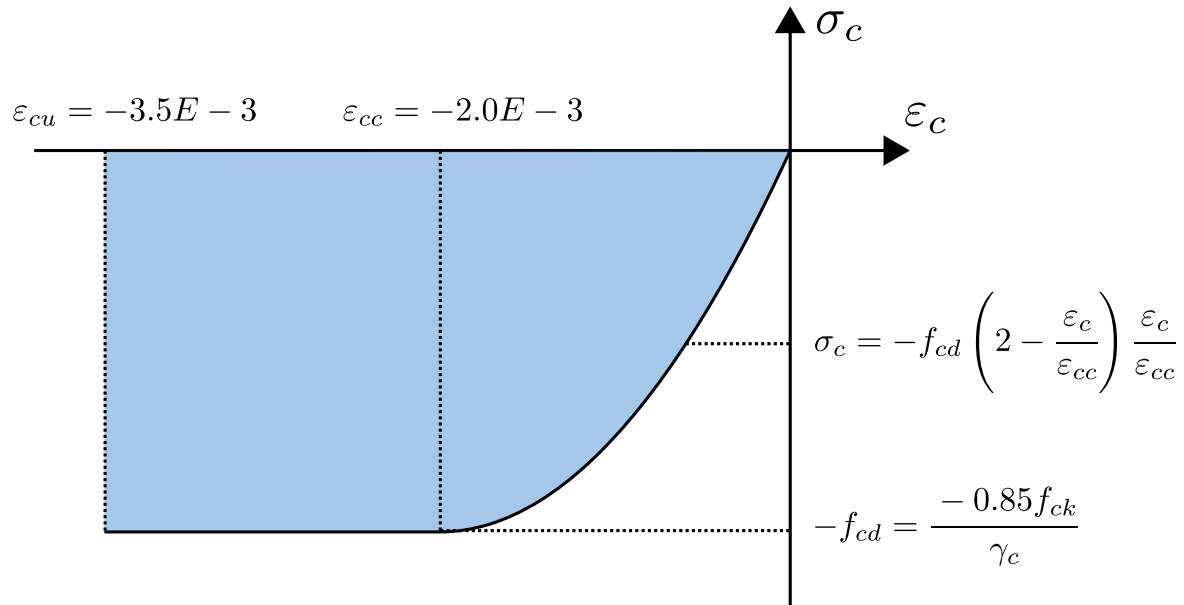
The relationship is defined as follows:

$$\varepsilon_c \geq 0 \qquad \sigma_c = 0$$

$$0 > \varepsilon_c \geq 0 \qquad \sigma_c = -f_{cd}\left(2 - \frac{\varepsilon_c}{\varepsilon_{cc}}\right)\frac{\varepsilon_c}{\varepsilon_{cc}}$$

$$\varepsilon_{cc} > \varepsilon_c \geq 0 \qquad \sigma_c = -f_{cd}$$

Remember that we have assumed compressive strain and stresses to be negative. The maximum strain is assumed to be ε_{cu}=-0.0035, and the yeld strain is assumed to be ε_{cc}=-0.002. The design yeld strength of the concrete is

$$f_{cd} = \frac{0.85 f_{ck}}{\gamma_c} = \frac{0.85 \cdot 25}{1.5} = 14.67\ MPa$$

Rebar steel

The design stress-strain diagram for the reinforcement bars is shown in the next figure:
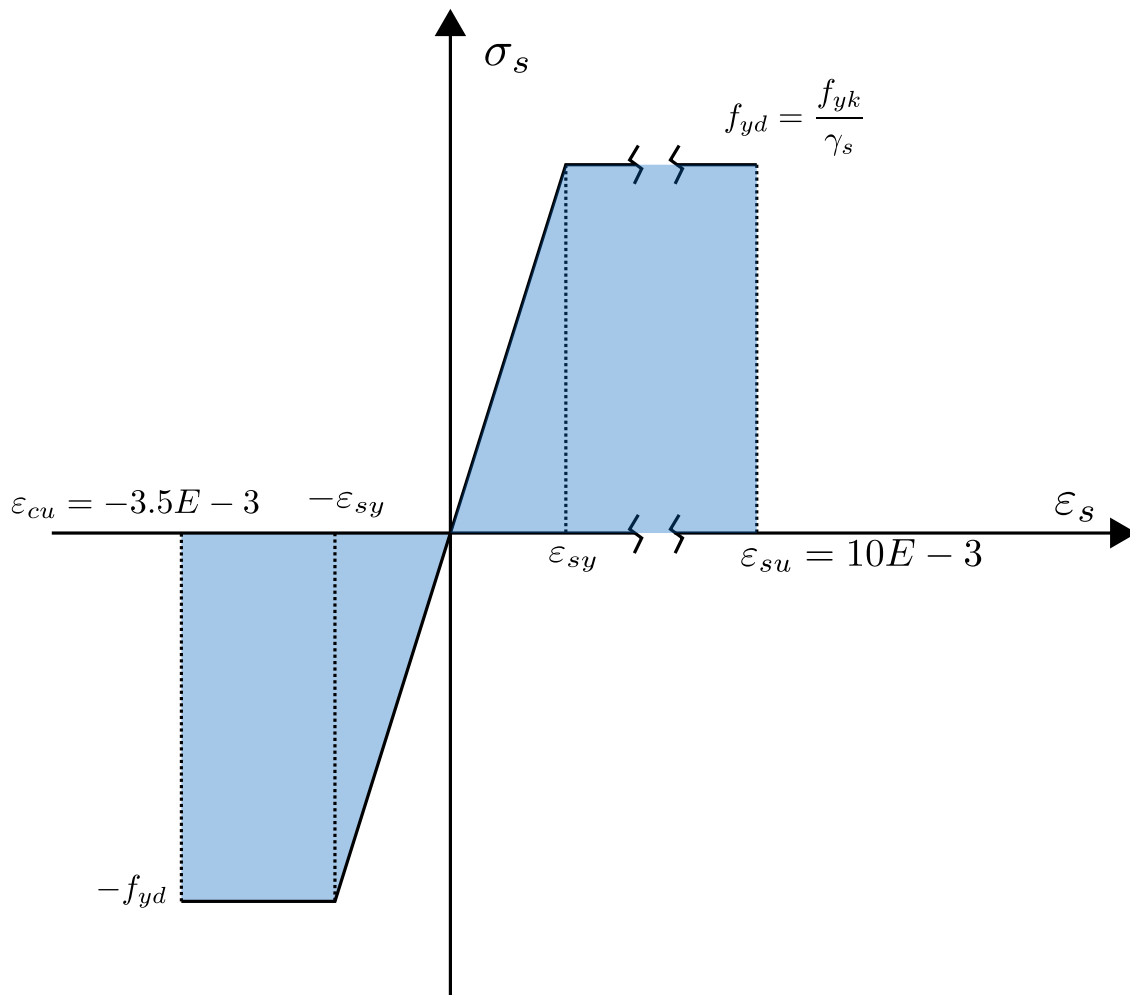
The branches of the diagram are defined as follows:

$$-\varepsilon_{cu} \leq \varepsilon_s \leq -\varepsilon_{sy} \qquad \sigma_s = -f_{yd}$$

$$-\varepsilon_{sy} < \varepsilon_s \leq \varepsilon_{sy} \qquad \sigma_s = \varepsilon_s E_s$$

$$\varepsilon_{sy} < \varepsilon_s \leq \varepsilon_{su} \qquad \sigma_s = f_{yd}$$

Notice how in the negative part of the diagram (compressive stresses) the maximum strain is ε_{cu} and not ε_{su}. The design young modulus is assumed to be E_s=200 GPa, and the maximum possible strain is limited at ε_{su}=0.01. The strain at failure is actually higher, but using the real value would allow for excessive deflections. The design yield stress is equal to

$$f_{yd} = \frac{f_{yk}}{\gamma_s} = \frac{450}{1.15} = 391.3 \; MPa$$

and the design yeld strain is

$$\varepsilon_{sy} = \frac{f_{yd}}{E_s} = \frac{391.3}{200000} = 0.00196$$

Strain domains

Because the concrete can't bear tensile stresses, we are facing a **non linear** problem. The portion of concrete that contributes to the resistance of the section depends on the depth of the neutral axis. This means that the resistance varies as the strain profile of the section changes. To plot the resistance domain we must therefore consider all the possible strain profiles that bring the section to failure, in accordance to the ULS design filosofy. For each profile there is an associated M-N couple that we can use to plot the resistance domain of the section.

We can define **five** different domains for the strain profile, as seen in the next figure:
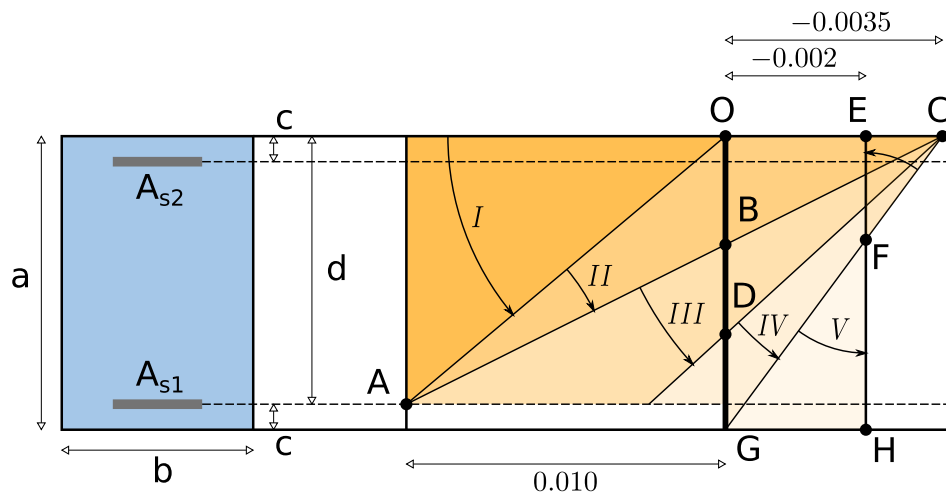
- **Domain** Ⅰ : The section is completely under tension, and the neutral axis is above it. The strain of the rebars at the bottom of the section is equal to $epsilon;_{su}$.
- **Domain** Ⅱ : The neutral axis is now inside the section and goes from **A** to **B**. In **B** the concrete reaches its maximum strain of -0.0035.
- **Domain** Ⅲ: The neutral axis now goes from **B** to **D**, where the bottom rebars reach their yeld strain.
- **Domain** Ⅳ: The neutral axis reaches the bottom of the section, which is now completely under compression.
- **Domain** Ⅴ : The neutral axis continues to lower below the section, and the strain profile rotates around **F**.

The strain diagram for each domain is described by a different set of equations, so they will be implemented in the code separately.

Python code

We will now begin to write the code necessary to implement the theory explained above. It is good practice to think about how you will structure your code *before* you start writing it, which involves a lot of standard pen and paper to write down your ideas. The final goal is to plot the limit resistance diagram of the section, which looks something like this:
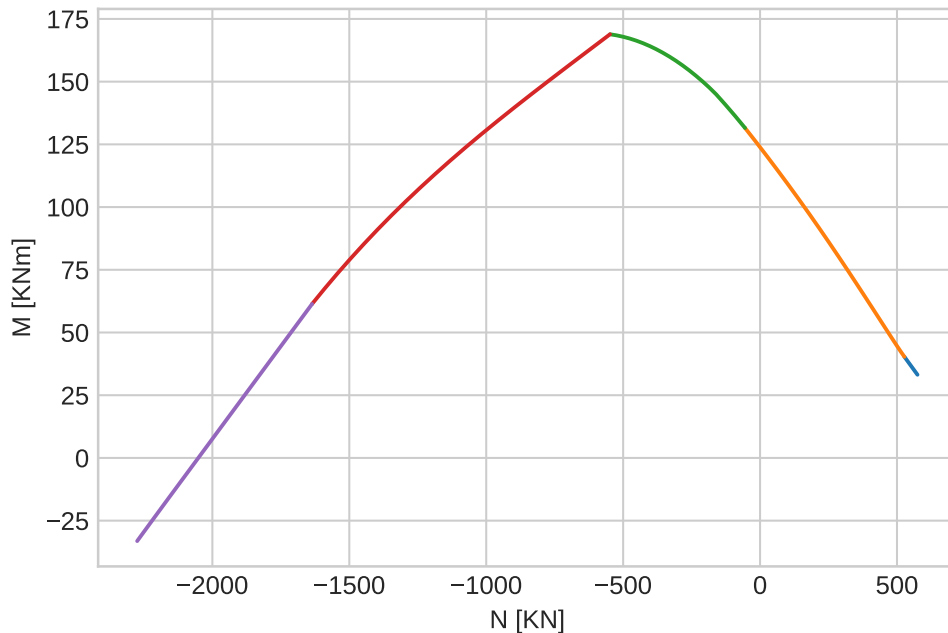
where the five different colors represent the five domains. Each point of the curve above represents a combination of axial force N and bending moment M that bring the section to failure. The steps to find this pair of values are as follows:

- Suppose a position of the neutral axis.
- In accordance to the domains listed above, find the strain distribution in the section. Because the section always remains flat, this distribution will be **linear**. Therefore, Knowing the position of the neutral axis and the strain in one point of the section is enough to find the distribution on the whole section.
- Using the stress-strain diagrams of the materials, find the stress distribution along the section.
- Integrate the stress distribution to find the axial force capacity N_{Rd} and the moment capacity M_{Rd}

Repeating these steps for all the possible positions of the neutral axis will result in the continuous curve seen in the previous figure. Having understood our problem, the next step is to code a solution. Here is how we will structure our code:

- Store the known quantities (such as dimensions, material properties, etc.) into variables.
- Write two functions (one for steel and one for concrete) that take an array of strain values as inputs. The output will be a corresponding array of stress values according to the stress-strain relations described previously.
- For each domain, consider the starting and ending positions of the neutral axis. Discretize this range of values with a fixed number of points, and store them in an array.
- For every discretized position of the neutral axis calculate the strain in the rebars and the concrete, and then use the functions defined previously to calculate the corresponding stresses.
- Integrate the stresses, find a pair of M-N values and store them in an array.

The first thing to do is to load all the necessary libraries. Create a new jupyter notebook, and import NumPy and matplotlib:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
```

Now we will input the section dimensions and the material properties. To keep the code more tidy we will use to separate cells. First, the section dimensions:

```
1  a=400 #mm, section height
2  b=300 #mm, section width
3  c=50 #mm, concrete cover
4  d=a-c
5  As1=1017 #mm^2, 4phi18, bottom rebar area
6  As2=452 #mm^2, 4phi12, top rebar area
```

Then the materials:

```
 1  #concrete
 2  fck=25 #MPa
 3  gamma_c=1.5
 4  fcd=0.85*fck/gamma_c #MPa
 5  eps_cc=-0.0020
 6  eps_cu=-0.0035
 7
 8  #steel
 9  fyk=450 #MPa
10  gamma_s=1.15
11  fyd=fyk/gamma_s #MPa
12  Es=200000 #MPa
13  eps_su=0.01
14  eps_y=fyd/Es
```

Concrete stress-strain diagram

We need a function that takes as input an array of strain values and outputs an array of stress values. As stated before, The relationship is described by the following equations:

$$\varepsilon_c \geq 0 \qquad \sigma_c = 0$$

$$0 > \varepsilon_c \geq 0 \qquad \sigma_c = -f_{cd}\left(2 - \frac{\varepsilon_c}{\varepsilon_{cc}}\right)\frac{\varepsilon_c}{\varepsilon_{cc}}$$

$$\varepsilon_{cc} > \varepsilon_c \geq 0 \qquad \sigma_c = -f_{cd}$$

The implementation goes as follows:

```
1   def rel_c(eps):
2       n=len(eps)
3       sig=np.zeros(n)
4       for i in range(n):
5           if eps[i]>=0:
6               sig[i]=0
7           elif 0>eps[i] and eps[i]>=eps_cc:
8               sig[i]=-fcd*(2-eps[i]/eps_cc)*eps[i]/eps_cc
9           elif eps_cc>eps[i] and eps[i]>=eps_cu:
10              sig[i]=-fcd
11          else:
12              print('invalid eps value')
13      return(sig)
```

In line 1 we create a new function called **rel_c**. The array of strain values that acts as input is called **eps** (short for "epsilon"). We use a *for* loop to cycle through each value of **eps**, implementing the equations using an IF statement. At each iteration we store the the result in an array called **sig** (short for "sigma"), which is the output of the function. If `eps[i]` is outside the bounds defined by the equations the function prints a warning. Should this happen, it means that there is something wrong with the strain values we are passing to the function.

Steel stress-strain diagram

The structure of the code is identical to what we have done for the concrete, except the equations to implement now are:

$$-\varepsilon_{cu} \leq \varepsilon_s \leq -\varepsilon_{sy} \qquad \sigma_s = -f_{yd}$$

$$-\varepsilon_{sy} < \varepsilon_s \leq \varepsilon_{sy} \qquad \sigma_s = \varepsilon_s E_s$$

$$\varepsilon_{sy} < \varepsilon_s \leq \varepsilon_{su} \qquad \sigma_s = f_{yd}$$

and the function is called **rel_s** instead of **rel_c**. The code goes as follows:

```
1   def rel_s(eps):
2       n=len(eps)
3       sig=np.zeros(n)
4       for i in range(n):
5           if -eps_su<=eps[i] and eps[i]<=-eps_y:
6               sig[i]=-fyd
7           elif -eps_y<eps[i] and eps[i]<=eps_y:
8               sig[i]=eps[i]*Es
9           elif eps_y<eps[i] and eps[i]<=eps_su:
10              sig[i]=fyd
11          else:
12              print('invalid eps value')
13      return(sig)
```

Domain I

We must decide a coordinate system for the position of the neutral axis. The origin will correspond to the top of the section, and the axis will be directed downward. For the first domain, then, the neutral axis will go from ∞ to 0. A general strain distribution is shown in the next figure:
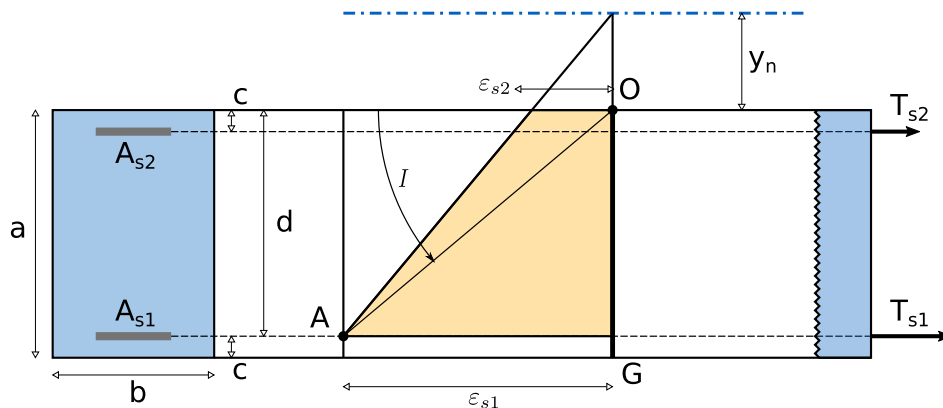
Because there is only positive strain (meaning that the whole section is under tension), the contribute of the concrete must be completely neglected. To calculate the M-N pair we must first find the strain of the top and bottom reinforcement bars. We know the position of the neutral axis y_n, and we know that the strain for the bottom rebars is equal to ε_{su}. Thus, the curvature of the strain diagram is equal to

$$\psi = \frac{\varepsilon_{su}}{d - y_n}$$

where $d-y_n$ is the total distance between the neutral axis and the bottom rebars. Remember that y_n is negative, that is why there is a minus sign in front of it. The strain of the top reinforcement bars is equal to

$$\varepsilon_{c2} = \psi(c - y_n)$$

The strain in the rebars generates a stress that multiplied by the area gives us the two forces T_{s1} and T_{s2}. The M-N pair can then be defined as follows:

$$N_{Rd} = T_{s1} + T_{s2}$$

$$M_{Rd} = T_{s1} \cdot \left(\frac{a}{2} - c \right) - T_{s2} \cdot \left(\frac{a}{2} - c \right)$$

N_{Rd} is applied to the centroid of the section, which is assumed to be in the middle of it. Since the reinforcement bars are not symmetrical the centroid is actually in a slightly different position, however tests show that the error caused by this assumption is negligible.

Let's implement all of the above equations:

```
1   n_step=20
2
3   yn_sup=-9999
4   yn_inf=0
5   yn=np.linspace(yn_sup, yn_inf, n_step)
6   psi=eps_su/(d-yn)
7   eps_s1=np.full(n_step, eps_su)
8   eps_s2=-psi*(yn-c)
9   sig_s1=rel_s(eps_s1)
10  sig_s2=rel_s(eps_s2)
11
12  Nrd_1=As1*sig_s1+As2*sig_s2
13  Mrd_1=As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
```

In line 1 we initialize a variable called **n_step** that defines the number of discretization points used to find the positions of the neutral axis. **yn_sup** and **yn_inf** store its starting and ending positions. For **yn_sup** we specify -9999, which is just a random high enough number that will serve in place of ∞. In line 5, using `np.linspace()` we create an array of evenly spaced values that go from yn_sup to yn_inf called **yn**. From line 6 to 8 we implement the equations that describe the strain distribution, and then we call **rel_s** twice to get the stresses for the top and bottom bars. Finally, in lines 12 and 13 we calculate and store all the M-N pairs in **Nrd_1** and **Mrd_1**. We now have two arrays that contain the resulting N_{Rd} and M_{Rd} for every position of the neutral axis contained in **yn**.

Domain II

The neutral axis is now inside the section, meaning that a part of it will be under tension and another under compression. This particular situation is shown in the next figure:
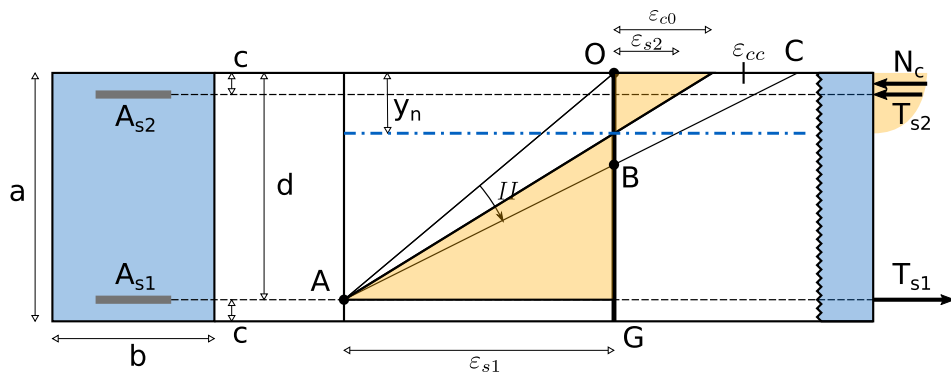
Point **C** corresponds to the maximum strain of the concrete ε_{cu}. Above the neutral axis the strain is negative, which translates to a negative stress distribution in the concrete. The integral of the stress distribution multiplied by the section width gives the total compression acting on the concrete N_c. The position of N_c along the y axis corresponds to the centroid of the shape of the stress distribution. The next figure illustrates this situation:
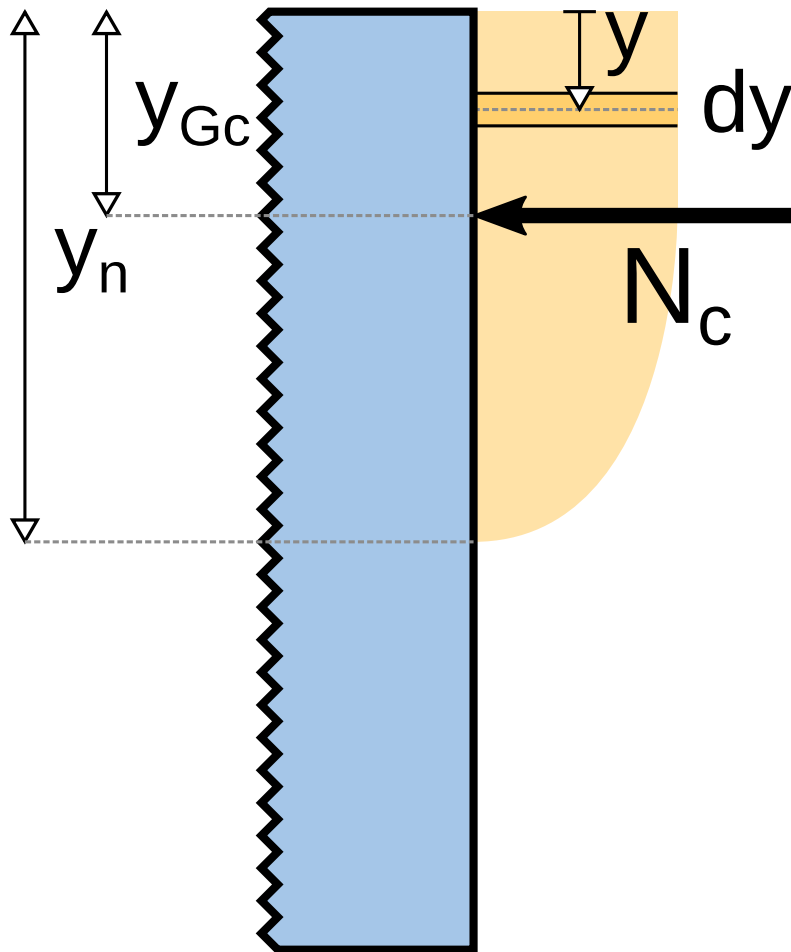
The yellow shape represents the stress in the concrete $\sigma_c(y)$. The value of N_c is given by:

$$N_c = b \int_0^{y_n} \sigma_c(y) \cdot dy$$

and the distance y_{Gc} is given by

$$y_{Gc} = \frac{\int_0^{y_n} y \cdot \sigma_c(y) \cdot dy}{\int_0^{y_n} \sigma_c(y) \cdot dy}$$

The starting position of the neutral axis is 0, and the ending position corresponds to a strain distribution where both the steel of the bottom bars and the concrete at the top of the section are at failure. This is what happens when the neutral axis is in **B**:

$$y_{n,inf} = \frac{d \cdot (-\varepsilon_{cu})}{\varepsilon_{su} + (-\varepsilon_{cu})}$$

The expressions for the strain in the bars remain the same as those for the first domain.

Now let's code all of this in a cell:

```
1   yn_sup=0
2   yn_inf=-d*eps_cu/(eps_su-eps_cu)
3   yn=np.linspace(yn_sup, yn_inf, n_step)
4   psi=eps_su/(d-yn)
5   eps_s1=np.full(n_step, eps_su)
6   eps_s2=-psi*(yn-c)
7   sig_s1=rel_s(eps_s1)
8   sig_s2=rel_s(eps_s2)
9
10  Nc=np.zeros(n_step)
11  Mc=np.zeros(n_step)
12  for i in range(n_step):
13      y=np.linspace(0, yn[i], 10)
14      eps_c=-(yn[i]-y)*psi[i]
15      eps_c=np.round(eps_c, 7)
16      sig_c=rel_c(eps_c)
17      Nc[i]=b*np.trapz(sig_c, y)
18      ygc=np.nan_to_num(np.trapz(sig_c*y, y)/np.trapz(sig_c, y))
19      Mc[i]=Nc[i]*(a/2-ygc)
20
21  Nrd_2=Nc+As1*sig_s1+As2*sig_s2
22  Mrd_2=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
```

The first eight lines are exactly the same as those written for the first domain. The only thing that changes are the starting and ending positions of the neutral axis. In lines 10 and 11 we initialize **Nc** and **Mc** as arrays of zeros. We then start a `for` loop that cycles through all the neutral axis positions stored in **yn**, and calculates Nc, ygc and Mc for each one of them. Let's take a closer look to what is written inside this `for` loop. The best way to calculate the integral is to discretyze the distance from 0 to y_n with a finite number of points, and then perform an approximate integration. In line 13 we create an array of 10 points that go from 0 to the current position of the neutral axis `yn[i]`. For each of these points we calculate the corresponding strain, according to this expression:

$$\varepsilon_c = -(y_n - y)\psi$$

and we store the results in an array called **eps_c**. In line 15 we round eps_c to five decimal points. The reason why we must do this is a bit cryptic. Because eps_c is the result of operations between floating point numbers, there is a small (and completely negligible) error introduced by the machine. The problem arises when we call **rel_c** and we pass a value that is *slightly* out of the bounds defined by the IF statement, such as 0.00350000001. This is exactly what happens when the neutral axis is in **B**, and the concrete has reached ε_{cu}. To correct this we simply round the values to an acceptable number of decimal poins, in this case 7.

Once the numbers in **eps_c** have been rounded, we can pass them to **rel_c** and obtain the stress distribution, that we store in **sig_c**. The next step is to integrate sig_c to calculate Nc. We do this using the trapezoidal rule, which numpy conveniently provides the user in the form of the function `np.trapz()`. The first argument contains the array we wish to integrate, and the second the position of each discretization point. In our case these are **sig_c** anf **y**. We use the same function to calculate **ygc**, with a little caviat: the first position of the neutral axis is 0, and in this configuration the integrals become 0 as well. This means that we are trying to calculate 0/0, which results in a `nan` value. The function `np.nan_to_num()` correct this problem by automatically changing any **nan** value to 0.

In lines 21 and 22 we calculate N_{Rd} and M_{Rd}. Remember that a bending moment that puts the bottom portion of the section under tension is *positive*. Mc however contains negative values, because compression stresses are considered to be negative. That is why there is a - sign in front of Mc, and the same goes for the bending moment given by the top bars.

Domain Ⅲ

The strain in the top-most fiber of the section is now fixed to ε_{cu}, and the neutral axis continues to lower until the strain in the bottom bars reaches ε_{sy}.
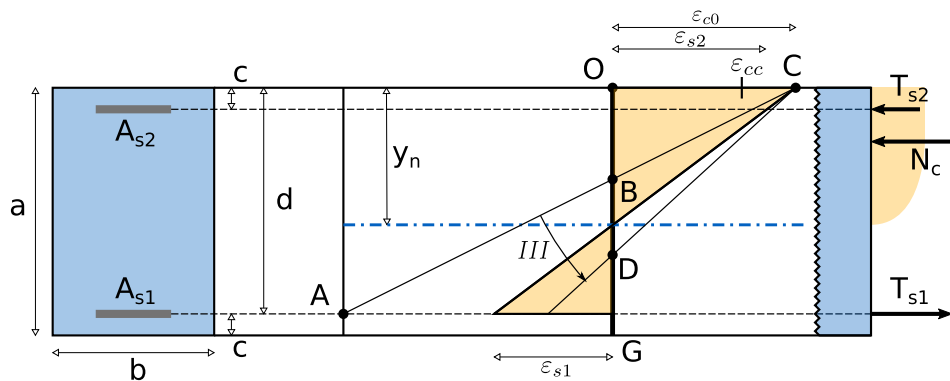
The neutral axis now goes from **B** to **C**, and the strain distribution rotates around **C**. The curvature is given by:

$$\psi = \frac{-\varepsilon_{cu}}{y_n}$$

and the strain in the bars is given by:

$$\varepsilon_{s1} = \psi \cdot (d - y_n)$$

$$\varepsilon_{s1} = -\psi \cdot (y_n - c)$$

The code is very similar to what we have already seen for the second domain:

```
 1  yn_sup=-d*eps_cu/(eps_su-eps_cu)
 2  yn_inf=-d*eps_cu/(eps_y-eps_cu)
 3  yn=np.linspace(yn_sup, yn_inf, n_step)
 4  psi=-eps_cu/yn
 5  eps_s1=-psi*(yn-d)
 6  eps_s2=-psi*(yn-c)
 7  eps_s1=np.round(eps_s1, 5)
 8  eps_s2=np.round(eps_s2, 5)
 9  sig_s1=rel_s(eps_s1)
10  sig_s2=rel_s(eps_s2)
11
12  Nc=np.zeros(n_step)
13  Mc=np.zeros(n_step)
14  for i in range(n_step):
15      y=np.linspace(0, yn[i], 10)
16      eps_c=-(yn[i]-y)*psi[i]
17      eps_c=np.round(eps_c, 7)
18      sig_c=rel_c(eps_c)
19      Nc[i]=b*np.trapz(sig_c, y)
20      ygc=np.trapz(sig_c*y, y)/np.trapz(sig_c, y)
21      Mc[i]=Nc[i]*(a/2-ygc)
22
23  Nrd_3=Nc+As1*sig_s1+As2*sig_s2
24  Mrd_3=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
```

Domain **IV**

The neutral axis now goes from **D** to the bottom of the section, while the strain in the top-most fiber remains ε_{cu}.
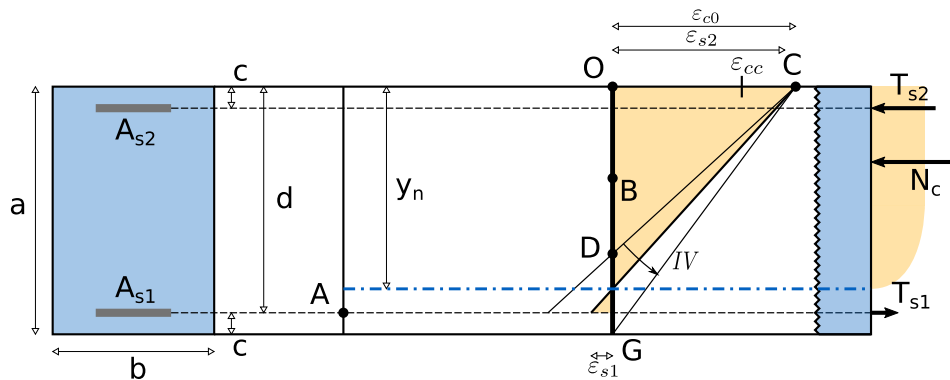
The expressions for the curvature and the strain remain the same as those for the third domain. Essentially the only things that change are the starting and ending positions of the neutral axis, and the rest of the code remains the same:

```
1   yn_sup=-d*eps_cu/(eps_y-eps_cu)
2   yn_inf=a
3   yn=np.linspace(yn_sup, yn_inf, n_step)
4   psi=-eps_cu/yn
5   eps_s1=-psi*(yn-d)
6   eps_s2=-psi*(yn-c)
7   sig_s1=rel_s(eps_s1)
8   sig_s2=rel_s(eps_s2)
9
10  Nc=np.zeros(n_step)
11  Mc=np.zeros(n_step)
12  for i in range(n_step):
13      y=np.linspace(0, yn[i], 10)
14      eps_c=-(yn[i]-y)*psi[i]
15      eps_c=np.round(eps_c, 7)
16      sig_c=rel_c(eps_c)
17      Nc[i]=b*np.trapz(sig_c, y)
18      ygc=np.trapz(sig_c*y, y)/np.trapz(sig_c, y)
19      Mc[i]=Nc[i]*(a/2-ygc)
20
21  Nrd_4=Nc+As1*sig_s1+As2*sig_s2
22  Mrd_4=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
```

Domain V

The neutral is now below the section, and it continues to lower until $y_n = \infty$. The strain distribution rotates around **F**, located at

$$t = \frac{3}{7}a$$

The curvature is given by:

$$\psi = \frac{-\varepsilon_{cc}}{y_n - t}$$

The strain in the bars can be calculated with the same expressions used for the fourth domain.
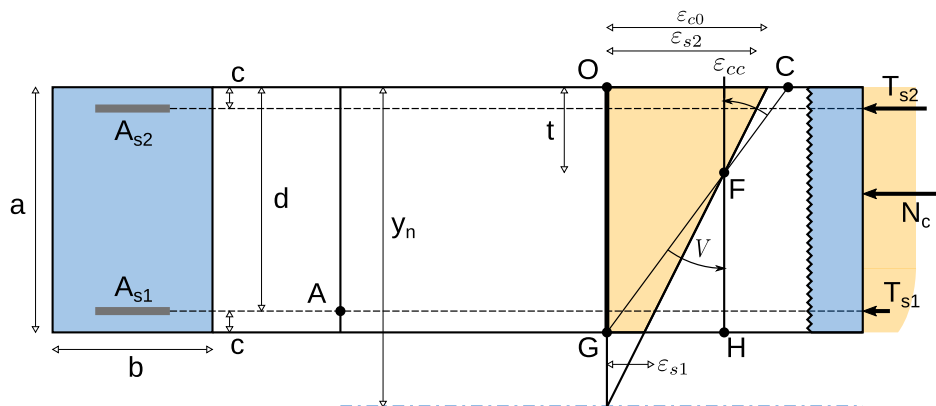
Apart from the starting and ending locations of the neutral axis, the code remains the same.

```python
1  yn_sup=a
2  yn_inf=a+9999
3  yn=np.linspace(yn_sup, yn_inf, n_step)
4  t=3/7*a
5  psi=-eps_cc/(yn-t)
6  eps_s1=-psi*(yn-d)
7  eps_s2=-psi*(yn-c)
8  sig_s1=rel_s(eps_s1)
9  sig_s2=rel_s(eps_s2)
10
11 Nc=np.zeros(n_step)
12 Mc=np.zeros(n_step)
13 for i in range(n_step):
14     y=np.linspace(0, a, 10)
15     eps_c=-(yn[i]-y)*psi[i]
16     eps_c=np.round(eps_c, 7)
17     sig_c=rel_c(eps_c)
18     Nc[i]=b*np.trapz(sig_c, y)
19     ygc=np.trapz(sig_c*y, y)/np.trapz(sig_c, y)
20     Mc[i]=Nc[i]*(a/2-ygc)
21
22 Nrd_5=Nc+As1*sig_s1+As2*sig_s2
23 Mrd_5=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
```

Plotting the results

We are finally ready to plot the limit resistance diagram of the section:

```
1  plt.style.use("seaborn-whitegrid")
2  fig, ax=plt.subplots(figsize=(6,4))
3  ax.plot(Nrd_1*1E-3, Mrd_1*1E-6)
4  ax.plot(Nrd_2*1E-3, Mrd_2*1E-6)
5  ax.plot(Nrd_3*1E-3, Mrd_3*1E-6)
6  ax.plot(Nrd_4*1E-3, Mrd_4*1E-6)
7  ax.plot(Nrd_5*1E-3, Mrd_5*1E-6)
8  ax.set_xlabel("N [KN]")
9  ax.set_ylabel("M [KNm]")
```
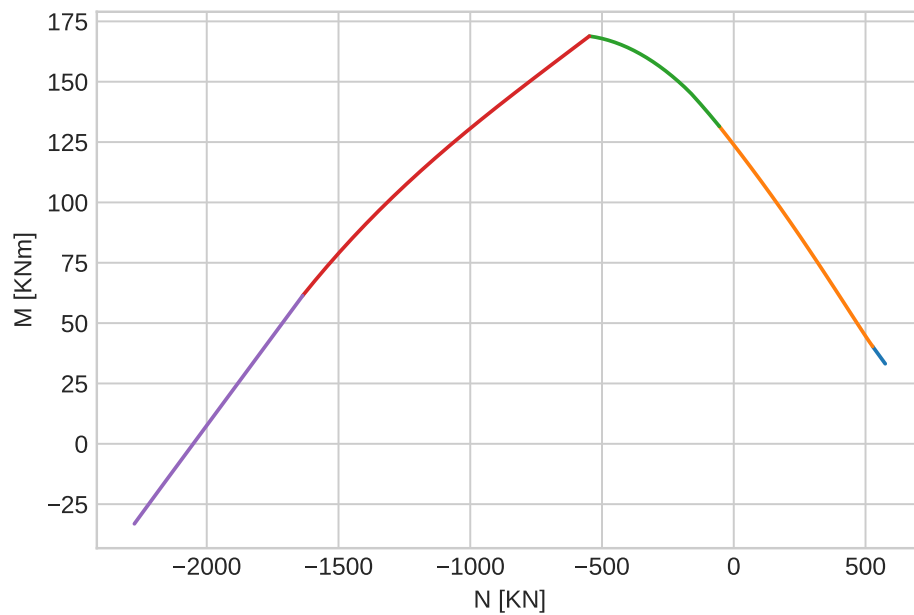
Conclusions

If you followed through this chapter carefully, you should now understand how to use **functions** to organize your code, and how to use *for* loops to perform operations on array elements. In the next chapter we will use the diagram that we have obtained to perform member verifications on a concrete beam. Obviously, the code only works for a rectangular section with top and bottom reinforcement bars, but expanding it shouldn't be difficult.

APPENDIX: saving the code in an external file

The notebook that we have written in this chapter is able to calculate the resistance diagram of a rectangular concrete section. However, it is rather impractical to utilize. Suppose that you had to plot the resistance diagram of ten different sections: you would need to change the input data of the notebook each time, surely there must be a better way to approach this problem. The solution is to wrap *everything* that we have written inside a single function. This function will take as inputs the dimensions of whatever cross-section we are considering, and return the resistance diagram. Then we will be able to call this function whenever we want. In order to do this, we need to create a **python script** that contains our function. A python script is just a text file that ends in **.py** that has some python code written inside.

Navigate to the folder where you have saved the notebook created in this chapter, and create a new file named **MN.py**. You can do this by right-clicking and selecting `new>text document`. This will create a .txt file that can then be renamed to MN.py. To edit its contents you can open it using notepad.

MN.py will contain a function called `getDomain(a,b,c,As1,As2,fck,fyk)` that takes the cross-section dimensions and the material properties as inputs. The block of code inside this function will be exactly the same as the code found in the cells of the notebook, so it is just a matter of copy-pasting the contents of each cell. Because The function will use numpy arrays, we need to import the library with the usual line of code `import numpy as np`, which will be placed in the first line of the file.

Here you have the whole function written in a single code block. You can copy-paste everything inside your file, or download a copy of MN.py from https://python4civil.weebly.com/m-n-diagram.html.

```python
import numpy as np

def getDomain(a,b,c,As1,As2,fck,fyk):
        '''
        Calculates Mrd for a given section and a
        given axial force.
        PARAMETERS:
        a =   section height (mm)
        b =   section width (mm)
        c =   concrete cover (mm)
        As1 = bottom rebar area (mm^2)
        As2 = top rebar area (mm^2)
        fck = concrete resistance (Mpa)
        RETURNS:
        (Nrd, Mrd) = resistance diagram (kNm)
        '''
        d=a-c

        #concrete
        gamma_c=1.5
        fcd=0.85*fck/gamma_c #MPa
        eps_cc=-0.0020
        eps_cu=-0.0035

        #steel
        gamma_s=1.15
        fyd=fyk/gamma_s #MPa
        Es=200000 #MPa
        eps_su=0.01
        eps_y=fyd/Es

        def rel_c(eps):
                # stress-strain for concrete
                n=len(eps)
                sig=np.zeros(n)
                for i in range(n):
                        if eps[i]>=0:
                                sig[i]=0
                        elif 0>eps[i] and eps[i]>=eps_cc:
                                sig[i]=-fcd*(2-eps[i]/eps_cc)*eps[i]/eps_cc
                        elif eps_cc>eps[i] and eps[i]>=eps_cu:
                                sig[i]=-fcd
                        else:
                                print('invalid eps value')
                return(sig)

        def rel_s(eps):
                # stress-strain for steel
                n=len(eps)
                sig=np.zeros(n)
                for i in range(n):
                        if -eps_su<=eps[i] and eps[i]<=-eps_y:
                                sig[i]=-fyd
                        elif -eps_y<eps[i] and eps[i]<=eps_y:
```

```python
                                sig[i]=eps[i]*Es
                        elif eps_y<eps[i] and eps[i]<=eps_su:
                                sig[i]=fyd
                        else:
                                print('invalid eps value')
                return(sig)

        n_step=50
        # DOMAIN I
        yn_sup=-9999
        yn_inf=0
        yn=np.linspace(yn_sup, yn_inf, n_step)
        psi=eps_su/(d-yn)
        eps_s1=np.full(n_step, eps_su)
        eps_s2=-psi*(yn-c)
        sig_s1=rel_s(eps_s1)
        sig_s2=rel_s(eps_s2)
        Nrd_1=As1*sig_s1+As2*sig_s2
        Mrd_1=As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)

        #DOMAIN II
        yn_sup=0.1
        yn_inf=-d*eps_cu/(eps_su-eps_cu)
        yn=np.linspace(yn_sup, yn_inf, n_step)
        psi=eps_su/(d-yn)
        eps_s1=np.full(n_step, eps_su)
        eps_s2=-psi*(yn-c)
        sig_s1=rel_s(eps_s1)
        sig_s2=rel_s(eps_s2)
        Nc=np.zeros(n_step)
        Mc=np.zeros(n_step)
        for i in range(n_step):
                y=np.linspace(0, yn[i], 50)
                eps_c=-(yn[i]-y)*psi[i]
                eps_c=np.round(eps_c, 7)
                sig_c=rel_c(eps_c)
                Nc[i]=b*np.trapz(sig_c, y)
                ygc=np.nan_to_num(np.trapz(sig_c*y, y)/np.trapz(sig_c, y))
                Mc[i]=Nc[i]*(a/2-ygc)
        Nrd_2=Nc+As1*sig_s1+As2*sig_s2
        Mrd_2=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)

        #DOMAIN III
        yn_sup=-d*eps_cu/(eps_su-eps_cu)
        yn_inf=-d*eps_cu/(eps_y-eps_cu)
        yn=np.linspace(yn_sup, yn_inf, n_step)
        psi=-eps_cu/yn
        eps_s1=-psi*(yn-d)
        eps_s2=-psi*(yn-c)
        eps_s1=np.round(eps_s1, 5)
        eps_s2=np.round(eps_s2, 5)
        sig_s1=rel_s(eps_s1)
        sig_s2=rel_s(eps_s2)
        Nc=np.zeros(n_step)
        Mc=np.zeros(n_step)
```

```python
110              for i in range(n_step):
111                      y=np.linspace(0, yn[i], 50)
112                      eps_c=-(yn[i]-y)*psi[i]
113                      eps_c=np.round(eps_c, 7)
114                      sig_c=rel_c(eps_c)
115                      Nc[i]=b*np.trapz(sig_c, y)
116                      ygc=np.trapz(sig_c*y, y)/np.trapz(sig_c, y)
117                      Mc[i]=Nc[i]*(a/2-ygc)
118          Nrd_3=Nc+As1*sig_s1+As2*sig_s2
119          Mrd_3=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
120
121          #DOMAIN IV
122          yn_sup=-d*eps_cu/(eps_y-eps_cu)
123          yn_inf=a
124          yn=np.linspace(yn_sup, yn_inf, n_step)
125          psi=-eps_cu/yn
126          eps_s1=-psi*(yn-d)
127          eps_s2=-psi*(yn-c)
128          sig_s1=rel_s(eps_s1)
129          sig_s2=rel_s(eps_s2)
130          Nc=np.zeros(n_step)
131          Mc=np.zeros(n_step)
132          for i in range(n_step):
133                      y=np.linspace(0, yn[i], 50)
134                      eps_c=-(yn[i]-y)*psi[i]
135                      eps_c=np.round(eps_c, 7)
136                      sig_c=rel_c(eps_c)
137                      Nc[i]=b*np.trapz(sig_c, y)
138                      ygc=np.trapz(sig_c*y, y)/np.trapz(sig_c, y)
139                      Mc[i]=Nc[i]*(a/2-ygc)
140          Nrd_4=Nc+As1*sig_s1+As2*sig_s2
141          Mrd_4=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
142
143          #DOMAIN IV
144          yn_sup=a
145          yn_inf=a+9999
146          yn=np.linspace(yn_sup, yn_inf, n_step)
147          t=3/7*a
148          psi=-eps_cc/(yn-t)
149          eps_s1=-psi*(yn-d)
150          eps_s2=-psi*(yn-c)
151          sig_s1=rel_s(eps_s1)
152          sig_s2=rel_s(eps_s2)
153          Nc=np.zeros(n_step)
154          Mc=np.zeros(n_step)
155          for i in range(n_step):
156                      y=np.linspace(0, a, 50)
157                      eps_c=-(yn[i]-y)*psi[i]
158                      eps_c=np.round(eps_c, 7)
159                      sig_c=rel_c(eps_c)
160                      Nc[i]=b*np.trapz(sig_c, y)
161                      ygc=np.trapz(sig_c*y, y)/np.trapz(sig_c, y)
162                      Mc[i]=Nc[i]*(a/2-ygc)
163          Nrd_5=Nc+As1*sig_s1+As2*sig_s2
164          Mrd_5=-Mc+As1*sig_s1*(a/2-c)-As2*sig_s2*(a/2-c)
```

```
165
166         Mrd=np.hstack((Mrd_1*1E-6,Mrd_2*1E-6, Mrd_3*1E-6,
167                         Mrd_4*1E-6, Mrd_5*1E-6))
168         Nrd=np.hstack((Nrd_1*1E-6,Nrd_2*1E-6, Nrd_3*1E-6,
169                             Nrd_4*1E-6, Nrd_5*1E-6))
170
171         return((Nrd, Mrd))
```

Designing a concrete beam

In this chapter we will design a two-span concrete beam in bending and shear, using the domain calculated in the previous chapter. The beam is exactly the same as the one in in the chapter about pandas, so we will import the bending moment and shear distributions as *.csv* files. This is a really simple example, so only two different load combinations will be considered. Still, two will be more than enough to explain how to find and plot the envelope and the maximum-minimum values. Then, we will find the critical sections to be considered, and calculate M_{Rd} for each one of them. Finally, we will move on to the shear verification of the beam. Throughout this chapter, verifications will be performed according to UNI EN 1992-1-1. Even if you are not familiar with european standards, you won't have any problem understanding this example. The goal is learning how to use python, not learning how to use the standards. By the end of this chapter, you won't have any problems applying what you have learned to your particular design environment.

Dimensions and loads

The load combinations are shown in the next figure:

COMBO 1

$q_1 = 25.4\ kN/m$

$q_2 = 15.2\ kN/m$

A   B   C

COMBO 2

$q_1 = 15.2\ kN/m$

$q_2 = 25.4\ kN/m$

A   B   C

$l_1 = 4\ m$ $l_2 = 5\ m$
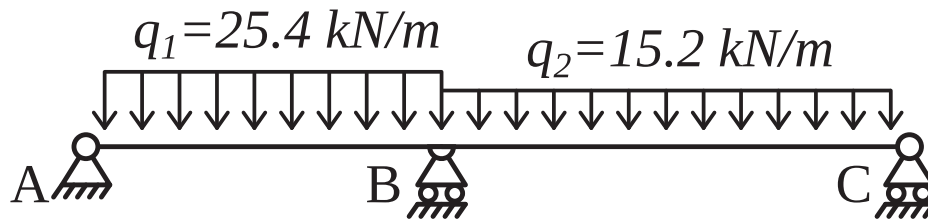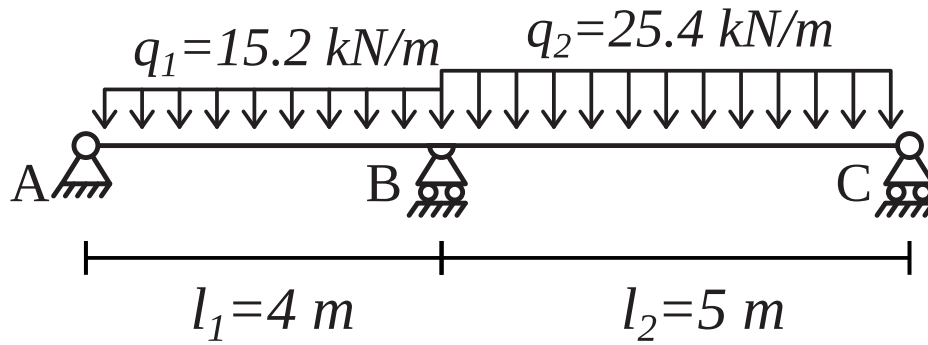
Importing the necessary libraries

In this example we will use **numpy**, **pandas** and **matplotlib**. In addition, we will also use the **MN.py** file created at the end of the previous chapter. Create a new folder containing MN.py and an empty jupyter notebook. In the first cell write:

```
1  import numpy as np
2  import pandas as pd
3  import matplotlib.pyplot as plt
4  import MN
```

the last line of code loads all the contents of **MN.py** in the notebook. Now we have access to the function `getDomain()`, and we use it by writing `MN.getDomain()`.

> **NOTE**
>
> If you modify the contents of **MN.py** after it has been loaded in the notebook, the changes won't register untill you restart the notebook's kernel.

Calculating the envelope of the distributions

Next, we need to import the bending moment and shear distributions. All the data we need is contained in a file called **beam.csv** that can be downloaded from https://python4civil.weebly.com/concrete-beam.html. The first column of the file contains the x coordinates of the beam, and the rest contain the values of the bending moment and shear at each point. In the next code block we use pandas to load the data and store it inside five numpy arrays:

```
1  data=pd.read_csv("beam.csv")
2  x=np.array(data.x)
3  M1=np.array(data.M1)  #kNm
4  M2=np.array(data.M2)  #kNm
5  V1=np.array(data.V1)  #kN
6  V2=np.array(data.V2)  #kN
```

The distributions are shown in the next figure:
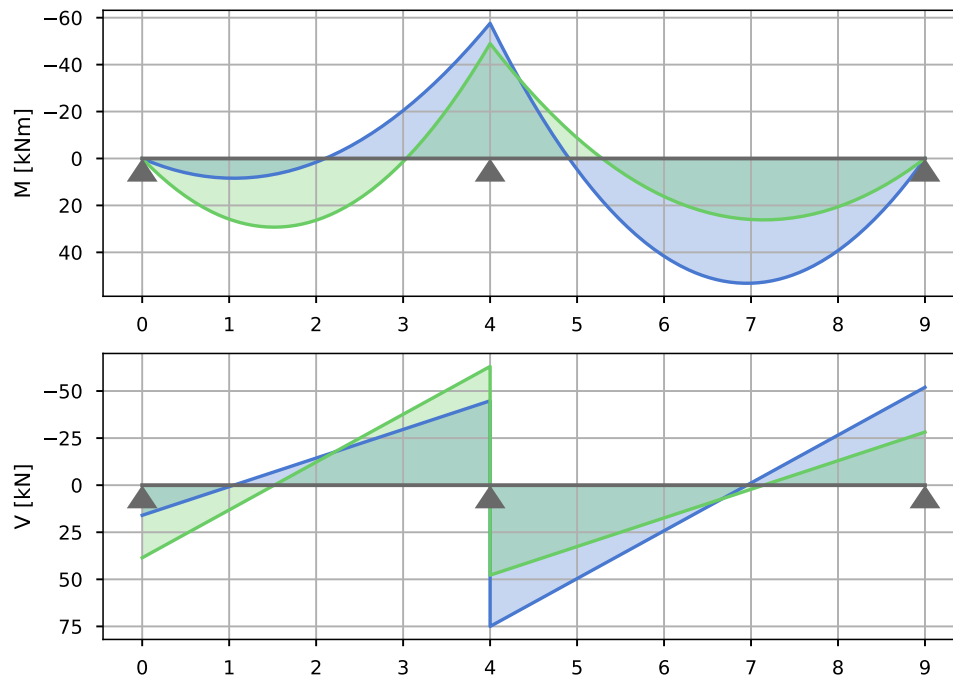
The distributions have been loaded and we are ready to calculate the envelope, meaning that we want to keep only the maximum values of the two load combinations. In order to do this we will create two separate vectors for the positive and negatives values, as you can see in the next code cell:

```
 1  M_neg=np.minimum(M1,M2)
 2  M_pos=np.maximum(M1,M2)
 3
 4  M_neg[M_neg>0]=0
 5  M_pos[M_pos<0]=0
 6
 7  V_neg=np.minimum(V1,V2)
 8  V_pos=np.maximum(V1,V2)
 9
10  V_neg[V_neg>0]=0
11  V_pos[V_pos<0]=0
```

By plotting the arrays that we have just created we obtain the following distributions_
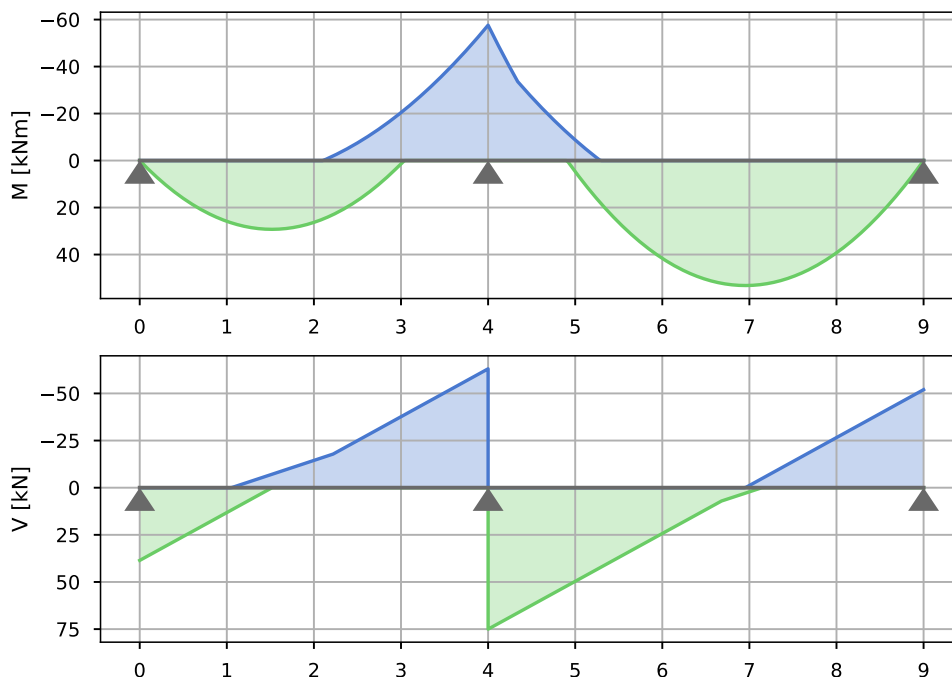
Extracting the maximum values from the envelope distributions

Now that we have the envelopes for both bending moment and shear, we need to extract the local maxima of the two distributions, so that we can start designing the critical sections. The problem is, numpy does not have a module for this specific task. Another well-known library called **scipy** would be able to do this, but we will code a solution ourselves, for the sake of learning. Let's define a function called **find_maxima** that takes an array as input, and outputs the local maxima and their locations in the array as a tuple:

```
1   def find_maxima(a):
2       maxima=[]
3       index=[]
4       n=len(a)
5       for i in range(n):
6           if i==0:
7               if a[0]>a[1]:
8                   maxima.append(a[0])
9                   index.append(i)
10          elif i==n-1:
11              if a[-1]>a[-2]:
12                  maxima.append(a[-1])
13                  index.append(i)
14          else:
15              if a[i]>=a[i-1] and a[i]>a[i+1]:
16                  maxima.append(a[i])
17                  index.append(i)
```

The code is pretty straight-forward. The input array **a** is scanned using a `for` loop. There is an IF-ELSE statement that checks wether the loop is currently at the beginning or the end of the array, to avoid indexing a wrong position. For each value of **i** we check wether the adjacent positions of `a[i]` hold a smaller value than `a[i]`. If that is the case, the value is stored inside **maxima** and **i** is stored inside **index**.

We can use this function to find the maxima of **M_pos** and **M_neg** by passing them as arguments:

```
1   M_pos_max, index_pos_M=find_maxima(M_pos)
2   M_neg_max, index_neg_M=find_maxima(abs(M_neg))
3   M_pos_max=np.array(M_pos_max)
4   M_neg_max=-np.array(M_neg_max)
5
6   V_pos_max, index_pos_V=find_maxima(V_pos)
7   V_neg_max, index_neg_V=find_maxima(abs(V_neg))
8   V_pos_max=np.array(V_pos_max)
9   V_neg_max=-np.array(V_neg_max)
```

Note that because **M_neg** and **V_neg** contain only negative values, we need to use `abs()` before passing them to `find_maxima()`. Then we can plot the location of the maxima using a scatter plot, obtaining something similar to the next figure:
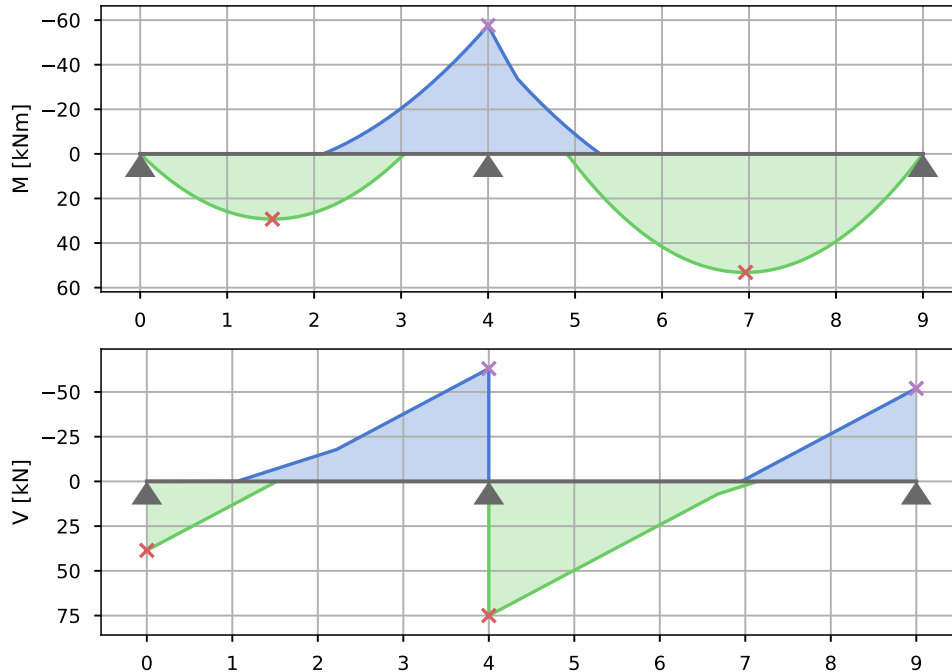
Now that we know the maximum values of M and V we can choose the materials and the section properties accordingly. Of course there is always some trial and error during this process, and often there are limiting factors on the materials that can be used and the dimensions of the beam. In this example we will keep things really simple, and consider three critical sections:

- **section 1:** located where the bending moment of the first span reaches its maximum
- **section 2:** located in the center support
- **section 3:** located where the bending moment of the second span reaches its maximum

The dimensions and the disposition of the reinforcement bars for each critical section are displayed in the next figure:

Section 1   **Section 2**   **Section 3**
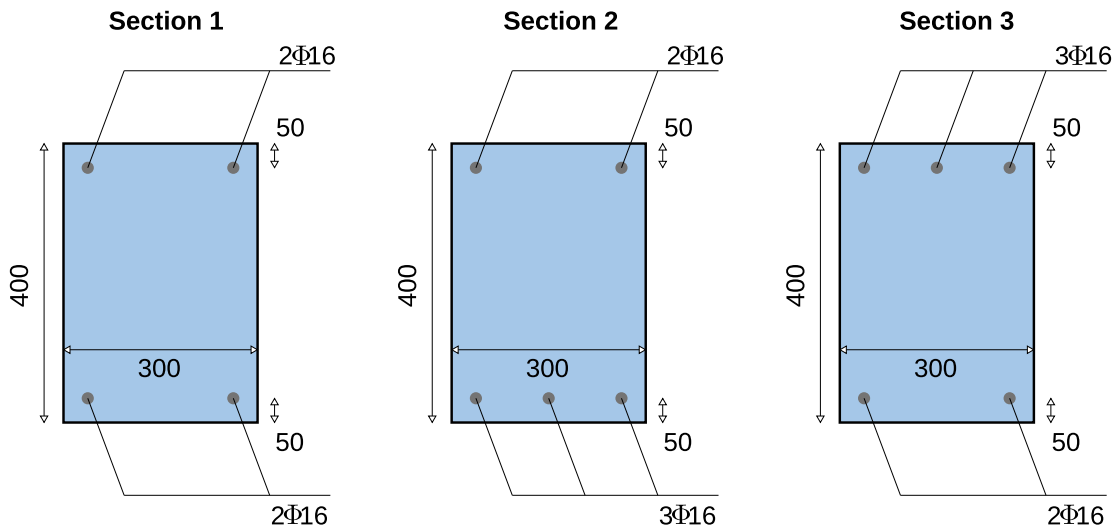
The class of the concrete is C25/30, and the steel material is B450C. As specified by the Eurocodes, γ_c=1.5 and γ_s=1.15. Then

$$f_{cd} = \frac{0.85 f_{ck}}{\gamma_c} = \frac{0.85 \cdot 25}{1.5} = 14.67 \ MPa$$

$$f_{yd} = \frac{f_{yk}}{\gamma_s} = \frac{450}{1.15} = 391.3 \ MPa$$

Let's store all the material and section properties into variables:

```
1   a=400  #mm
2   b=300  #mm
3   c=50  #mm
4   d=a-c  #mm
5   z=0.9*d  #mm
6   Abar=201.062  #mm^2
7
8   fck=25  #N/mm^2
9   fyk=450  #N/mm^2
10  gamma_s=1.15
11  gamma_c=1.5
12  fcd=0.85*fck/gamma_c  #N/mm^2
```

where:

- **a** is the height of the section
- **b** is the width of the section
- **c** is the concrete cover
- **d** is the effective section
- **z** is the inner lever arm
- **Abar** is the total area of a $\Phi 16$ rebar

Bending moment resistance of the sections

It is now time to use the functions contained in MN.py. To be more specific, we will use `MN.getDomain()` to retrieve the limit resistance diagrams for each one of the sections. The function needs the following parameters:

1. Section height (mm)
2. Section width (mm)
3. Bottom rebar area (mm^2)
4. Top rebar area (mm^2)
5. Concrete compressive strength f_{ck} (N/mm^2)
6. Steel yeld strength f_{yk} (N/mm^2)

The output is a tuple containing N_{Rd} and M_{Rd}. We don't actually need the whole domain, we only want to know M_{Rd} for $N_{Rd}=0$. We will use the numpy function **interp** to interpolate the domain and find $M_{Rd}(N_{Rd}=0)$:

```
1  Nrd_tot, Mrd_tot=MN.getDomain(a, b, c, 2*Abar, 2*Abar, fck, fyk)
2  Mrd1=np.interp(0, Nrd_tot[::-1], Mrd_tot[::-1])
3  Nrd_tot, Mrd_tot=MN.getDomain(a, b, c, 3*Abar, 2*Abar, fck, fyk)
4  Mrd3=np.interp(0, Nrd_tot[::-1], Mrd_tot[::-1])
5  Mrd2=-Mrd3
```

In line 1 we pass **getDomain** all the necessary arguments to find the resistance domain for **section 1**. The function returns two vectors: **Nrd_tot** and **Mrd_tot**. These are the *x* and *y* coordinates of the domain. In line 2 using `np.interp` we interpolate these two vectors to get the value of M_{Rd} for $N_{Rd}=0$. The way this function works is very simple: the first parameter is the value for which we wish to interpolate, the second is a vector that contains the positions along the x axis and the third is a vector containing the value corresponding to each position. In this case Nrd_tot will act as the x axis and Mrd_tot will act as the y axis. There is one important caviat however: **interp** only works if the second parameter is a vector whose values are sorted in ascending order (like the horizontal axis of a cartesian graph). The output of **getDomain**, however, gives us a Nrd_tot vector sorted in *descending* order (because compressive stresses are assumed to be negative, refer to the previous chapter for more informations on this topic). Basically, we need to revers the order of Nrd_tot and Mrd_tot for **interp** to work, and we do this with `[::-1]`. In line 3 we re-use Nrd_tot and Mrd_tot to store the

resistance domain for section 3, and then interpolate for Nrd_tot=0. Finally, because the bending resistance of section 2 to negative bending moments is the same as section 3, there is no need to call **getDomain** again. So in line 5 we simply invert the sign of Mrd3 to get Mrd2.

> **NOTE**
>
> The function **getDomain** calculates the resistance domain for *positive* bending moments. To calculate the resistance to negative bending moments we simply need to switch the positions of the reinforcement bars, and treat the negative bending moment as if it was positive.

Section verifications

Now that we know the bending moment resistances for each of the three section we can compare them with the design bending moments calculated before. A simple `if-else` statement will suffice for this example:

```python
 1  if M_pos_max[0]<Mrd1:
 2      print("sec. 1: VERIFIED")
 3  else:
 4      print("sec. 1: NOT VERIFIED")
 5  if M_neg_max[0]>Mrd2:
 6      print("sec. 2: VERIFIED")
 7  else:
 8      print("sec. 2: NOT VERIFIED")
 9  if M_pos_max[1]<Mrd3:
10      print("sec. 3: VERIFIED")
11  else:
12      print("sec. 3: NOT VERIFIED")
```

```
sec. 1: VERIFIED
sec. 2: VERIFIED
sec. 3: VERIFIED
```

The problem with this approach is that it is not very scalable, because it only works with three sections. If we had a beam with three spans instead of just two we would need to add more lines. Ideally everything should be as automated as possible, but for now this method will be enough.

Shear verification

Section 6.2.2 of EN 1992-1-1 provides the formulas used for shear member verification. For elements that do not require design shear reinforcement the value of the shear resistance $V_{Rd,c}$ is given by:

$$V_{Rd,c} = \left[C_{Rd,c} k \left(100 \rho_l f_{ck} \right)^{1/3} + k_1 \sigma_{cp} \right] b_w d$$

with a minimum of

$$V_{Rd,c} = (v_{min} + k_1 \sigma_{cp}) b_w d$$

where:

- f_{ck} is in MPa
- $1+(200/d)^{0.5}$ with d in mm
- $\rho_l = A_{sl}/(b_w d) \leq 0.02$
- S_{sl} is the area of the rebarsunder tension
- b_w is the width of the cross-section
- $\sigma_{cp} = N_{Ed}/A_c < 0.2 f_{cd}$ MPa
- N_{Rd} is the axial force in the cross-section
- $V_{Rd,c}$ is in newton.

If the design shear V_{Ed} is less than $V_{Rd,c}$, then there is no need to carry out the verifications, as long as the minimum requirements specified in section 9.2.2 of eurocode 2 are met.

The first thing to do in jupyter is to find the maximum shear value:

```
1   Vmax=max(max(V_pos_max), max(abs(V_neg_max)))
2   print("Vmax = \%d"\%(Vmax)+" kN")

Vmax = 75 kN
```

Then it is just a matter of implementing the formulas:

```
 1  bw=b
 2  k=1+(200/d)**0.5
 3  rho_l=max(3*Abar/(bw*d), 0.02)
 4  sig_cp=0
 5  Ac=b*a
 6  CRd_c=0.18/gamma_c
 7  k1=0.15
 8  nu_min=0.035*k**1.5*fck**0.5
 9
10  Vrd_c=max((CRd_c*k*(100*rho_l*fck)**(1/3)+k1*sig_cp)*bw*d,
11             (nu_min+k1*sig_cp)*bw*d)
12  print("Vrd_c = \%d"\%(Vrd_c*0.001)+" kN")
```

```
Vrd_c = 81 kN
```

Because Vmax<Vrd_c we can stop here. Eurocode 2 then prescribes a minimum concrete reinforcement ratio of

$$\rho_{w,min} = \left(0.08\sqrt{f_{ck}}/f_{yk}\right)$$

that needs to be observed when choosing the area of the shear reinforcement. We will use Φ8 vertical links with a step of 250 mm, which give us the following results:

```
 1  Asw=100 #mm^2, 2phi8
 2  s=250 #mm, step
 3  rho_w=Asw/(s*bw)
 4  rho_w_min=(0.08*(fck)**0.5)/fyk
 5  print("rho_w= \%.6f"\%(rho_w))
 6  print("rho_w_min= \%.6f"\%(rho_w_min))
```

```
rho_w= 0.001333
rho_w_min= 0.000889
```

note that **s** has been chosen according to

$$s_{l,max} = 0.75d(1 + cot\alpha)$$

Conclusions

Hopefully this chapter has given you a solid starting point to begin developing your own member verifications in python. Later in this book you will learn how to use **pandas** to create more scalable applications, and also how to typeset the results in various useful formats.

Designing a steel column

The goal of this chapter is to design a steel column subject to vertical and horizontal loads. We will use **pandas** manage different load combinations, and conveniently store the results in a table. The verifications will be carried out in accodance to the eurocodes, but like in the previous chapter you don't need to be familiar with the european standards to follow along.

As always, let's create a new jupyter notebook and import all the necessary libraries:

```python
import numpy as np
import pandas as pd
```

We will consider a column fixed at the base and free at the top. The next figure shows the position of the loads:
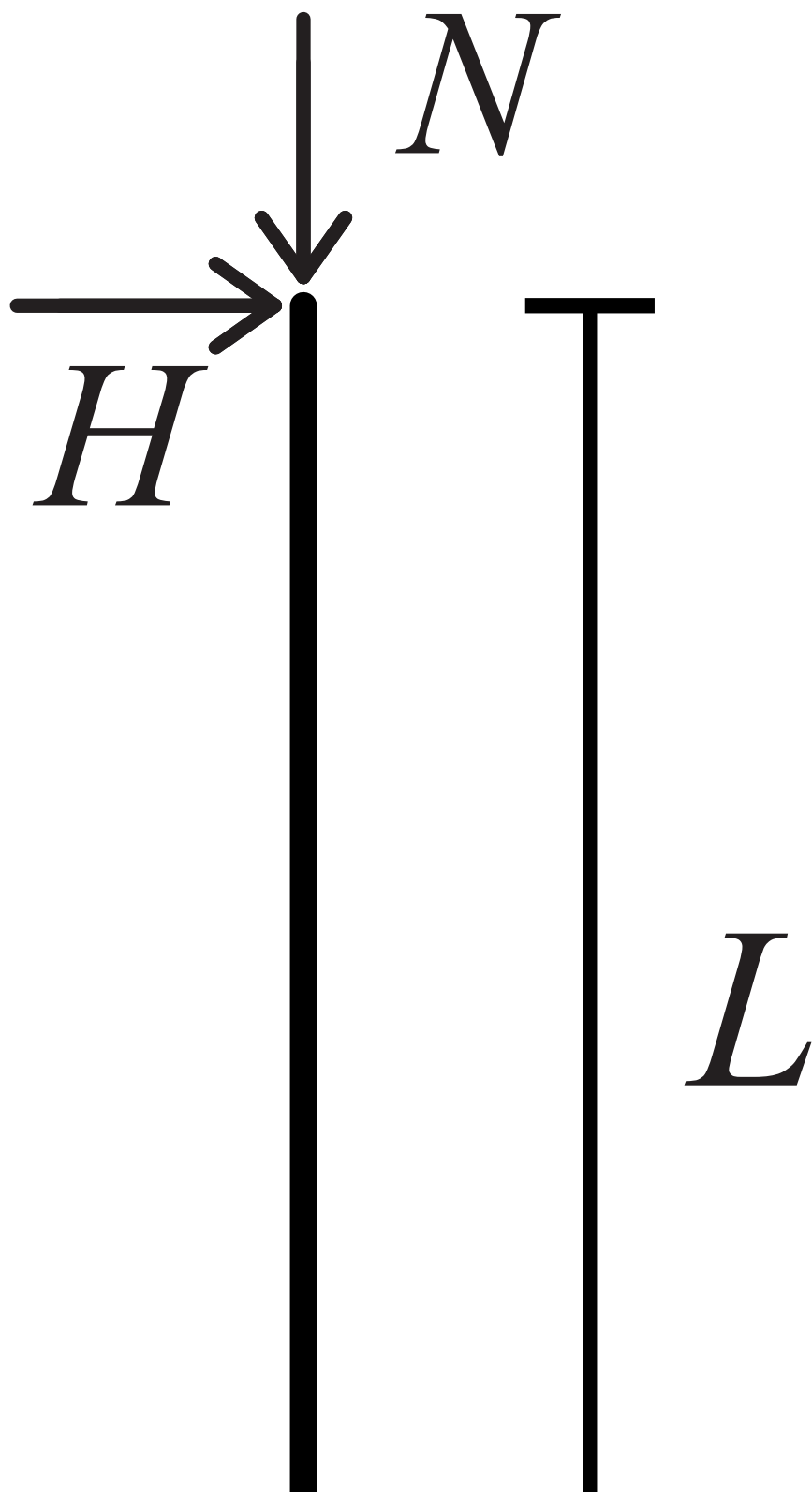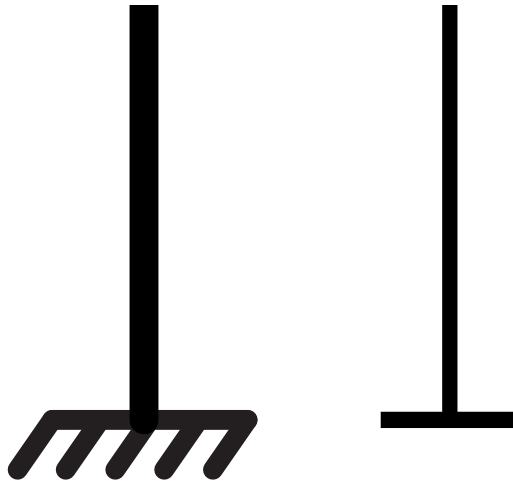
In this example we will consider **L** to be **7.5 m**, and the loads to be grouped in three different load combinations:

- **Combo 1:** N_{Ed}=300 kN, H_{Ed}=30 kN
- **Combo 2:** N_{Ed}=500 kN, H_{Ed}=10 kN
- **Combo 3:** N_{Ed}=250 kN, H_{Ed}=40 kN

Now let's input all these values in the notebook. We will use a variable named *L* to store the length, and a dataframe named *loads* to store the loads combinations:

```
1  L=7.5*10**3 #mm
2  combo1=pd.Series({"Ned":300, "Hed":30}, name="Combo1")
3  combo2=pd.Series({"Ned":500, "Hed":10}, name="Combo2")
4  combo3=pd.Series({"Ned":250, "Hed":40}, name="Combo3")
5
6  loads=pd.DataFrame([combo1, combo2, combo3])
7  loads["Med"]=loads.Hed*L*10**(-3)
8  loads
```

	Ned	Hed	Med
Combo1	300	30	225.0
Combo2	500	10	75.0
Combo3	250	40	300.0

Instead of creating **loads** all in one go, we first create three series called **combo1**, **combo2** and **combo3** to make the code cleaner. The maximum bending moment will be equal to H*L, so in line 7 we create a new column called **Med** and store the result of this calculation in it.

Materials

In this example we will use a standard **S355** as the steel grade. Let's store all the properties that we need into variables:

```
1  fyk=355  #N/mm^2
2  E=200000  #Mpa
3  G = 81000  #N/mm^2
```

Cross-section

Now let's pick a standard hot-rolled section and store its properties into correspondent variables. We will consider a **HE300B**, with the following characteristics:

```python
1   # HE300B
2   h=300  #mm (section height)
3   b=300  #mm (section width)
4   tw=11  #mm (web width)
5   tf=19  #mm (flange width)
6   A=161.3*10**2  #mm^2 (total area)
7   Iy=25170*10**4  #mm^4 (second moment of area, y axis)
8   iy=13*10  #mm (radius of gyration, y axis)
9   Wy=1678*10**3  #mm^3 (elastic section modulus, y axis)
10
11  Iz=8563*10**4  #mm^4 (second moment of area, z axis)
12  iz=7.58*10  #mm (radius of gyration, z axis)
13  Wz=570.9*10**3  #mm^3 (elastic section modulus, z axis)
14
15  It=185*10**4  #mm^4 (torsional constant)
16  Iw=(Iz*(h-tf)**2)/4  #mm^6 (warping constant)
17
18  fyk=355  #N/mm^2
19  E=200000  #Mpa
20  G = 81000  #N/mm^2
```

Where the y is the strong axis and z is the weak axis. A more practical approach would be to load the section properties from an external database, such as an excel or csv document.

Column buckling

Steel columns almost never undergo cross-section failure, because the buckling resistance of the member is usually much smaller than the resistance of the cross section. This example makes no exception, so for the sake of brevity we will skip section verifications.

According to Eurocode 3 (ENV 1993-1-1), Members subjected to combined bending and axial compression should satisfy the following equations:

$$\frac{N_{Ed}}{\dfrac{\chi_y N_{Rk}}{\gamma_{M1}}} + k_{yy}\frac{M_{y,Ed}}{\chi_{LT}\dfrac{M_{y,Rk}}{\gamma_{M1}}} + k_{yz}\frac{M_{z,Ed}}{\dfrac{M_{z,Rk}}{\gamma_{M1}}} \leq 1$$

$$\frac{N_{Ed}}{\dfrac{\chi_y N_{Rk}}{\gamma_{M1}}} + k_{zy}\frac{M_{y,Ed}}{\chi_{LT}\dfrac{M_{y,Rk}}{\gamma_{M1}}} + k_{zz}\frac{M_{z,Ed}}{\dfrac{M_{z,Rk}}{\gamma_{M1}}} \leq 1$$

where

- χ_y, χ_z are the reduction factors due to flexural buckling
- χ_{LT} is the reduction factor due to lateral torsional buckling
- $N_{Rk}=A\cdot f_y$
- $M_{y,Rk}=W_{el,y}\cdot f_y$
- $M_{z,Rk}=W_{el,z}\cdot f_y$
- k_{yy}, k_{yz}, k_{zy} and k_{zz} are the interaction factors

In our case $M_{z,Ed}=0$, so we can remove it from the formulas:

$$\frac{N_{Ed}}{\dfrac{\chi_y N_{Rk}}{\gamma_{M1}}} + k_{yy}\frac{M_{y,Ed}}{\chi_{LT}\dfrac{M_{y,Rk}}{\gamma_{M1}}} \leq 1$$

$$\frac{N_{Ed}}{\dfrac{\chi_y N_{Rk}}{\gamma_{M1}}} + k_{zy}\frac{M_{y,Ed}}{\chi_{LT}\dfrac{M_{y,Rk}}{\gamma_{M1}}} \leq 1$$

The reduction factor χ_y can be calculated as follows:

$$\chi_y = \frac{1}{\left(\phi + \sqrt{(\phi^2 - \bar{\lambda}^2)}\right)}$$

$$\phi = 0.5 + \left[1 + \alpha\left(\bar{\lambda} - 0.2\right) + \bar{\lambda}^2\right]$$

$$\bar{\lambda} = \sqrt{\frac{A f_y}{N_{cr}}} = \left(\frac{L_{cr}}{i}\right)\left(\frac{1}{\lambda_1}\right)$$

In our case:

$$L_{cr} = 2L$$

$$\lambda_1 = \pi\sqrt{\frac{E}{f_y}} = 93.9\epsilon$$

$$\epsilon = \sqrt{\frac{235}{f_y}}$$

The buckling curve to be considered is a function of h/b, as stated in table 6.2 of the eurocode. In our case h/b=1 and the buckling curve is **a**, which gives us an imperfection factor α=0.21.

Having understood the procedure, let's implement the formulas to calculate χ_y:

```
 1  alpha_y = 0.21
 2  Lcr_y = 2*L
 3  lambda_y = (Lcr_y/iy)*(1/lambda_1)
 4  phi_y = 0.5*(1+alpha_y*(lambda_y-0.2)+lambda_y**2)
 5  chi_y = 1/(phi_y+np.sqrt(phi_y**2-lambda_y**2))
 6  print("alpha_y= %.2f"%(alpha_y))
 7  print("Lcr_y= %.2f"%(Lcr_y))
 8  print("lambda_y= %.2f"%(lambda_y))
 9  print("phi_y= %.2f"%(phi_y))
10  print("chi_y= %.7f"%(chi_y))
```

```
alpha_y= 0.21
Lcr_y= 15000.00
lambda_y= 1.55
phi_y= 1.84
chi_y= 0.3531319
```

Next we need to calculate χ_{LT}, according to what specified in section 6.3.2.2 of Eurocode 3.

$$\chi_{LT} = \frac{1}{\phi_{LT} + \sqrt{\phi_{LT} - \beta\bar{\lambda}_{LT}^2}}$$

$$\phi_{LT} = 0.5\left[1 + \alpha_{LT}\left(\bar{\lambda}_{LT} - \bar{\lambda}_{LT,0}\right) + \beta\bar{\lambda}_{LT}^2\right]$$

$$\bar{\lambda}_{LT} = \sqrt{\frac{W_{pl,y}f_y}{M_{cr}}}$$

M_{cr} is the elastic critical moment for lateral-torsional buckling. Eurocode 3 does not provide a specific formula to calculate it, leaving to the engineers the choice of a suitable expression. The previous version of the eurocode suggested this formula:

$$M_{cr} = C_1\frac{\pi^2 EI_z}{(kL)^2}\sqrt{\left(\frac{k}{k_w}\right)\frac{I_w}{I_z} + \frac{(kL)^2 GI_t}{\pi^2 EI_z}}$$

where k_w and k depend on the types of constraints, and C_1 on the bending moment distribution. For a column (or beam) with one fixed end the literature suggests k_w=0.7, k=0.7 and C_1=0.7.

α_{LT} depends on the buckling curve, and in our case is equal to 0.34. As for β, the recommended value is 0.75.

Now let's implement all these formulas in a notebook cell:

```python
k=0.7
kw=0.7
C1 = 2.092
alpha_lt = 0.34
lambda_lt_0 = 0.4
beta = 0.75

Mcr = C1*np.pi**2*E*Iz/((k*L)**2)*np.sqrt((k/kw)*Iw/(Iz)\
                +(k*L)**2*G*It/(np.pi**2*E*Iz))*1e-6

lambda_lt = np.sqrt(Wy*fyk/(Mcr*1e6))
phi_lt = 0.5*(1+alpha_lt*(lambda_lt-lambda_lt_0)+beta*lambda_lt**2)
chi_lt = 1/(phi_lt+np.sqrt(phi_lt**2-beta*lambda_lt**2))

print("Mcr= %.2f"%(Mcr))
print("lambda_lt= %.2f"%(lambda_lt))
print("phi_lt= %.2f"%(phi_lt))
print("chi_lt= %.2f"%(chi_lt))
```

```
Mcr= 2696.44
lambda_lt= 0.47
phi_lt= 0.59
chi_lt= 0.97
```

Now let's calculate k_{yy} and k_{zy}, according to what specified in Annex B of Eurocode 3. For class 1 and 2 sections the following expressions apply:

$$k_{yy} = C_{my}1.8$$

$$k_{yz} = 0.6k_{zz}$$

$$k_{zy} = 1$$

$$k_{zz} = C_{mz}2.4$$

$$C_{my} = 0.6$$

Implementing the formulas we obtain:

```
 1  k=0.7
 2  kw=0.7
 3  C1 = 2.092
 4  alpha_lt = 0.34
 5  lambda_lt_0 = 0.4
 6  beta = 0.75
 7
 8  Mcr = C1*np.pi**2*E*Iz/((k*L)**2)*np.sqrt((k/kw)*Iw/(Iz)\
 9                      +(k*L)**2*G*It/(np.pi**2*E*Iz))*1e-6
10
11  lambda_lt = np.sqrt(Wy*fyk/(Mcr*1e6))
12  phi_lt = 0.5*(1+alpha_lt*(lambda_lt-lambda_lt_0)+beta*lambda_lt**2)
13  chi_lt = 1/(phi_lt+np.sqrt(phi_lt**2-beta*lambda_lt**2))
14
15  print("Mcr= %.2f"%(Mcr))
16  print("lambda_lt= %.2f"%(lambda_lt))
17  print("phi_lt= %.2f"%(phi_lt))
18  print("chi_lt= %.2f"%(chi_lt))
```
```
kyy= 1.08
kzz= 1.44
kyz= 0.86
kzy= 1.00\\
```

The last elements that we need are N_{Rk} and $M_{Rk,y}$, given by the following expressions:

$$N_{Rk} = Af_y$$

$$M_{rk,y} = W_{pl,y}f_y$$

$$M_{rk,z} = W_{pl,z}f_y$$

Implementing the formulas we obtain:

```
1  Nrk = A*fyk*0.001    # kN
2  Mrk_y = Wy*fyk*1e-6  # kNm
3
4  print("Nrk= %.2f"%(Nrk))
5  print("Mrk_y= %.2f"%(Mrk_y))
```
```
Nrk= 5726.15
Mrk_y= 595.69
```

Member verification

We finally have everything that we need to implement the verification formulas:

$$\frac{N_{Ed}}{\dfrac{\chi_y N_{Rk}}{\gamma_{M1}}} + k_{yy}\frac{M_{y,Ed}}{\chi_{LT}\dfrac{M_{y,Rk}}{\gamma_{M1}}} \leq 1$$

$$\frac{N_{Ed}}{\dfrac{\chi_y N_{Rk}}{\gamma_{M1}}} + k_{zy}\frac{M_{y,Ed}}{\chi_{LT}\dfrac{M_{y,Rk}}{\gamma_{M1}}} \leq 1$$

Because we have conveniently stored the loads inside the columns of a dataframe, we can write the formulas once and obtain the results for every load combination:

```
1  gamma_m1 = 1.05
2  results=pd.DataFrame()
3  results["check1"] = loads.Ned/(chi_y*Nrk/gamma_m1)\
4              +kyy*(loads.Med)/(chi_lt*Mrk_y/gamma_m1)
5  results["check2"] = loads.Ned/(chi_y*Nrk/gamma_m1)\
6              +kzy*(loads.Med)/(chi_lt*Mrk_y/gamma_m1)
7  results
```

	check1	check2
Combo1	0.596262	0.563633
Combo2	0.406460	0.395584
Combo3	0.717126	0.673621

In line 1 we specify the partial factor for resistance of members to instability γ_{M1}, and then in line 2 we create an empty dataframe called **results**. Then we apply the formulas above storing the results in two new columns **check1** and **check2**. All the values are smaller than 1, so the verification can be considered satisfied.

Conclusions

This chapter is meant to be an introduction on how to use pandas to store data and perform calculations. The key aspect here is **scalability**: No matter how many load combinations there are, the code that you have written will always work. In the next chapter you will use pandas more extensively, and learn how to export a notebook in LaTeX format.

Exporting in Latex

In this chapter we will create a Jupyter notebook that can be exported in LaTeX format. In order to have some formulas to work with we will consider a simply supported steel beam, subject to an uniform load **q**.

Before we start writing any code, however, we need to expand the capabilities of Jupyter using third-party **extensions**. To be more specific, we will install a set of extensions called **nbextensions**, that will allow as to use Latex syntax in markdown cells and also to hide the code we write.

> **NOTE**
>
> Even if you are not familiar with LaTeX, you can still follow along the code written in this chapter. The code works anyway, whether you have installed **nbextensions** or not.

Installing nbextensions

If you followed the Anaconda installation at the beginning of the book, your computer should also have installed **anaconda prompt** by default. Anaconda prompt is a command line interface that allows the user to install additional packages using the **pip** or **conda** package managers. We will use conda to install **nbextensions**. Open anaconda navigator, and type:

```
conda install -c conda-forge jupyter\_contrib\_nbextensions
```

This will install nbextensions from the anaconda repositories, so you need to be connected to the internet for the command to run.

> **NOTE**
>
> If you have trouble installing nbextensions, you can find plenty of tutorials online. I suggest you take a look to the official nbextensions installation guide, which can be found here:
> https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/install.html

Now when you start jupyter notebook you can access the configuration page by writing "nbextensions" in place of "tree" in the address bar. The configuration page looks like this:
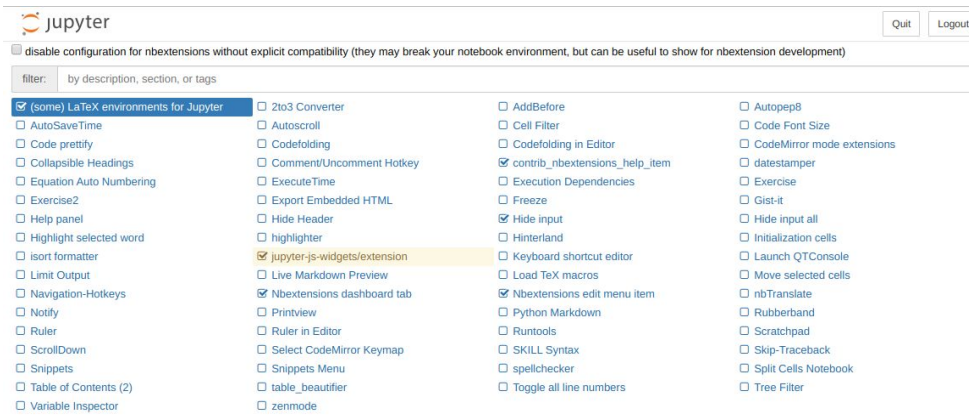
From here you can activate the extensions that you need. In order to be able to write latex syntax inside markdown cell we will need the extension called **(some) LaTeX environments for Jupyter**, and in order to hide code cells we will also need the **hide input** extension.

Now create a new notebook. You should see some new icons in the toolbar:
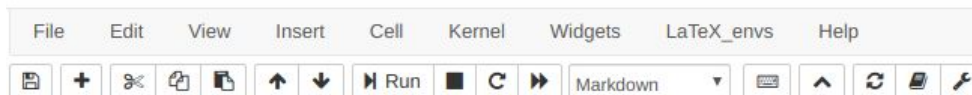
The button with the ^ symbol is used to hide the input of code cells. Now that LaTeX_envs is enabled we can also use some basic LaTeX environments inside markdown cells, like **section** and **itemize** for example, or math formulas using dollar signs.

Formatting the output of cells in LaTeX

If the final goal is to output the notebook as a LaTeX document (.tex), then we need a way to display the outputs of code cells accordingly. A way to do this is using the **Latex** module contained in `IPython.display`:

```python
from IPython.display import Latex
a=15
b=7
res=a+b
display(Latex("$a+b=%d+%d=%d$"%(a,b,res)))
```

$a + b = 15 + 7 = 22$

As you can see the output of the cell is now beautifully rendered in LaTeX.

Converting dataframes to LaTeX tables

Pandas gives the user the possibility to convert any dataframe into a LaTeX table, using `pandas.DataFrame.to\_latex()`. Here is an example:

```python
import pandas as pd
df=pd.DataFrame({"A":[3,2,1,6], "B":[4,5,9,7]}, index=[0,1,2,3])
display(Latex(df.to_latex(bold_rows=True, column_format="lll")))
```

```
\begin{tabular}{lll}
\toprule
{} &  A &  B \\
\midrule
\textbf{0} &  3 &  4 \\
\textbf{1} &  2 &  5 \\
\textbf{2} &  1 &  9 \\
\textbf{3} &  6 &  7 \\
\bottomrule
\end{tabular}
```

The function accepts a lot of parameters to customize the appearence of the table. Setting **bold_rows** to *True*, for example, will make print the row index in bold font. Another useful parameter is **column_format**, to which you can pass a string such as "rlc" to specify the alignment of the columns ("r"=*right*, "l"=*left*, "c"=*center*). Notice how in order to display the LaTeX syntax correctly we need to wrap `df.to_latex()` inside `Latex()`. The table however is still very basic: it does not have a caption, and it does not look very good. An easy solution is simply to add more LaTeX code around the output of `Latex(df.to_latex())`. For example, we could use the **table** environment like this:

```python
display(Latex(
    "\\begin{table}[H]\n\centering\n"\
    +df.to_latex(bold_rows=True, column_format='lll')\
    +"\caption{My Table}\n \end{table}"))
```

```
\begin{table}[H]
\centering
\begin{tabular}{lll}
\toprule
{} &  A &  B \\
\midrule
\textbf{0} &  3 &  4 \\
\textbf{1} &  2 &  5 \\
\textbf{2} &  1 &  9 \\
\textbf{3} &  6 &  7 \\
\bottomrule
\end{tabular}
\caption{My Table}
 \end{table}
```

The output of `df.to\_latex()` is just a simple string, so we can concatenate it with any other strings using +. In this case we are wrapping it inside `\begin{table} \end{table}` as well as specifying a caption using `caption{My Table}`. Because strings are one-line sequences of characters, we need to tell **Latex()** where the linebreaks are using `\n`.

The final result is a code cell that has a LaTeX table in the output. Unfortunately jupyter is not able to render it, so all we see is the raw LaTeX code. After you have exported the notebook, however, this raw code will become part of a .tex file, and thus you will able to render it using any LaTeX editor of your choice.

Calculating the deflection of a steel beam

In this example we will consider three different simply supported steel beams of length L_1, L_2 and L_3 subjected to three different uniform loads q_1, q_2, and q_3. For each beam we will calculate the maximum deflection.

- **Beam 1** L_1=5m, q_1=20 KN/m
- **Beam 2** L_2=6m, q_2=18 KN/m
- **Beam 3** L_3=6.5m, q_3=13 KN/m

All three beams will have the same cross section and material. We will use a standard **IPE220**, with a second moment of inertia around the strong axis of 2772 mm^4. The young modulus of the steel is 200000 Mpa. As always, the first step is to save all the relevant data into variables or dataframes:

```python
import pandas as pd
import numpy as np

Iy=2772*10**4  #mm^4
E=200000  #Mpa

loads=np.array([20, 18, 13])  #kN/m
lengths=np.array([5, 6, 6.5])  #m
```

To demonstrate the capability of exporting **matplotlib** plots in LaTeX, let's plot the moment distribution of the first beam:

```python
import matplotlib.pyplot as plt

x=np.linspace(0,lengths[0], 51)
M1=loads[0]*lengths[0]*x/2-(loads[0]*x**2)/2
fig, ax= plt.subplots()
ax.invert_yaxis()
ax.plot(M1)
plt.savefig("plot1.pdf")
plt.show()
```
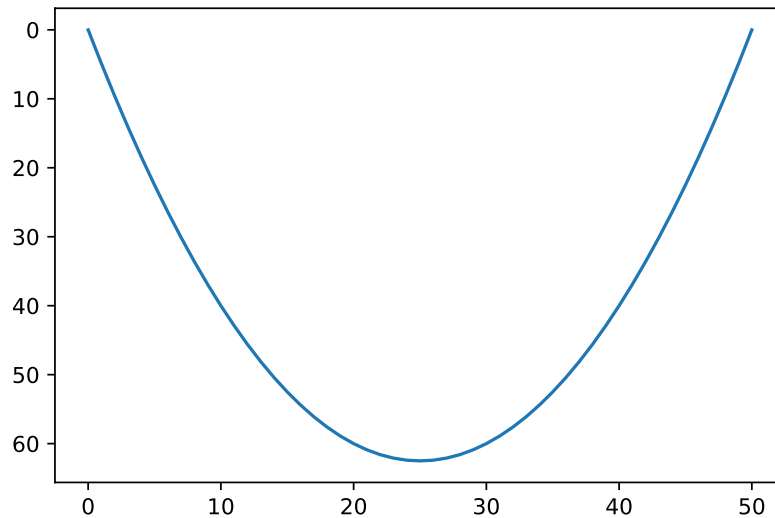
Now we need to calculate the maximum deflection of the beam, using the well-known formula from elastic theory:

$$w_{max} = \frac{5}{384} \frac{qL^4}{EI}$$

All the calculations will be performed inside a dataframe called **results**. Each row represents one of the beams, and we will add more columns as we calculate new quantities. First, we create the dataframe specifying only the index:

```
1   results=pd.DataFrame(index=(["Beam 1", "Beam 2", "Beam 3"]))
```

Then we can start adding columns. Let's add the loads and the lengths first:

```
1   results["l"]=lengths
2   results["q"]=loads
3
4   display(Latex(
5       "\\begin{table}[H]\n\centering\n"\
6       +results.to_latex(bold_rows=True, column_format='lll')\
7       +"\caption{My Table}\n \end{table}"))
```

```
\begin{table}[H]
\centering
\begin{tabular}{lll}
\toprule
{} &    l &   q \\
\midrule
\textbf{Beam 1} &  5.0 &  20 \\
\textbf{Beam 2} &  6.0 &  18 \\
\textbf{Beam 3} &  6.5 &  13 \\
\bottomrule
\end{tabular}
\caption{My Table}
 \end{table}
```

Now we can use the existing columns to perform calculations and store the results in new ones. If you have used a spreadsheet before you should be familiar with this type of workflow, and will find that pandas dataframes behave very similarly to Excel tables. Let's finish this example by calculating the maximum moments and deflections:

```
1  results["$M_{max}$"]=(results.q*results.l**2)/8
2  results["$w_{max}$"]=(5/384)*(results.q*results.l**4)/(E*Iy*0.001**3)
3
4  display(Latex(
5      "\\begin{table}[H]\n\centering\n"\
6      +results.to_latex(bold_rows=True, column_format='lllll', escape=False)\
7      +"\caption{My Table}\n \end{table}"))
```

```
\begin{table}[H]
\centering
\begin{tabular}{lllll}
\toprule
{} &    l &   q &  $M_{max}$ &   $w_{max}$ \\
\midrule
\textbf{Beam 1} &  5.0 &  20 &   62.50000 &   0.029358 \\
\textbf{Beam 2} &  6.0 &  18 &   81.00000 &   0.054789 \\
\textbf{Beam 3} &  6.5 &  13 &   68.65625 &   0.054502 \\
\bottomrule
\end{tabular}
\caption{My Table}
\end{table}
```

There are a few things to note here. First, the headers of the new columns that have been created contain latex syntax. It will look ugly in jupyter, but once the notebook has been exported in latex format the table will be rendered correctly. Also, there is a new argument passed to `to_latex()` which is **escape**, set to *False*. This prevents pandas from escaping special latex characters in column names.

Exporting the notebook

Now we come to the final step of this chapter, which is exporting the notebook that we have just created. The tool that we will use is called **nbconvert**. It is a command line interface that comes pre-installed with jupyter and can be accessed with anaconda prompt. Maybe you have never used a command line interface, so let's first explain what it is and what you can do with it.

Using a command line interface

A command line interface is just a different way of accessing files in running programs. Open **anaconda prompt**, type `ls` and hit enter. You should now see a list of all the files and folders contained in the path specified in the command line. To enter one of the folders, simply type `cd` followed by the folder's name. If you type `cd ..` you will enter the parent folder. Using `cd}` you can go anywhere in your computer, but sometimes it is faster to navigate to a folder directly. In order to do this simply type `cd` followed by the **absolute path** of the folder you want to access. Remember to use forward slashes, and to enclose folder names that have spaces in them using quotation marks.

Using nbconvert

Now let's move on to using nbconvert to export the notebook into a .tex file. Using anaconda prompt, navigate to the folder where the notebook is saved. Once there, type

```
jupyter nbconvert --to latex --template article notebook\_name.ipynb
```

where *notebook_name* is the name of the notebook you want to convert. Press enter, and you should now see that there is a new file in the folder called *notebook_name.tex*. If you open it and run it using your trusted LaTeX editor, you will most likely get an error. This is because the notebook that we have created in this chapter contains latex code that requires additional packages, such as **float** in order to display the tables correctly. We will solve this problem later: first let's explain the command that we have typed to obtain this result.

Think of **jupyter nbconvert** as a function call to which you can pass various parameters. **--to** specifies the format of the output. By default this is set to html. **--template** specifies the template that nbconvert will use while performing the conversion. A template is basically a set of rules that the converter has to follow. In this case we set it to **article**, which produces a decent enough latex document, but the images don't display correctly and all the code is visible. The solution is to use a custom template that recognizes when the code is hidden and renders the images better.

Improving the output

Teaching how to write a custom template is beyond the scope of this book, so we will use one that has already been created. Go to https://python4civil.weebly.com/exporting-in-latex.html

and download temp.tplx, and place it in the same folder of your jupyter notebook. Hide every code input in the notebook, and save it. Then in anaconda prompt run

```
jupyter nbconvert --to latex --template temp.tplx notebook\_name.ipynb
```

This will update the existing latex file already present in the folder. Now open it with your favourite LaTeX editor and run it: the result should be a nicely typesetted document without any python code showing, and with nice images and tables.

Conclusions

Most engineers know how to use LaTeX to create professional documents. Being able to perform calculations on a jupyter notebook and then to export everything in a .tex file can save a lot of time, and automate something that normally would take hours. I invite you to try to customize the template file that was used to export the notebook so that it better suits your needs.

Wrapping up

You have finally reached the end of this book. You should now have a solid understanding of the python programming language, and how to use it effectively. If this was the first time you approached programming, congratulations, the hardest part of the learning process is now behind you. If instead you were already a seasoned programmer, hopefully you still found some valuable knowledge in here. In any case, you now have the tools necessary to solve a large variety of problems. Remember that the more you code the easier it gets, so don't stop here!