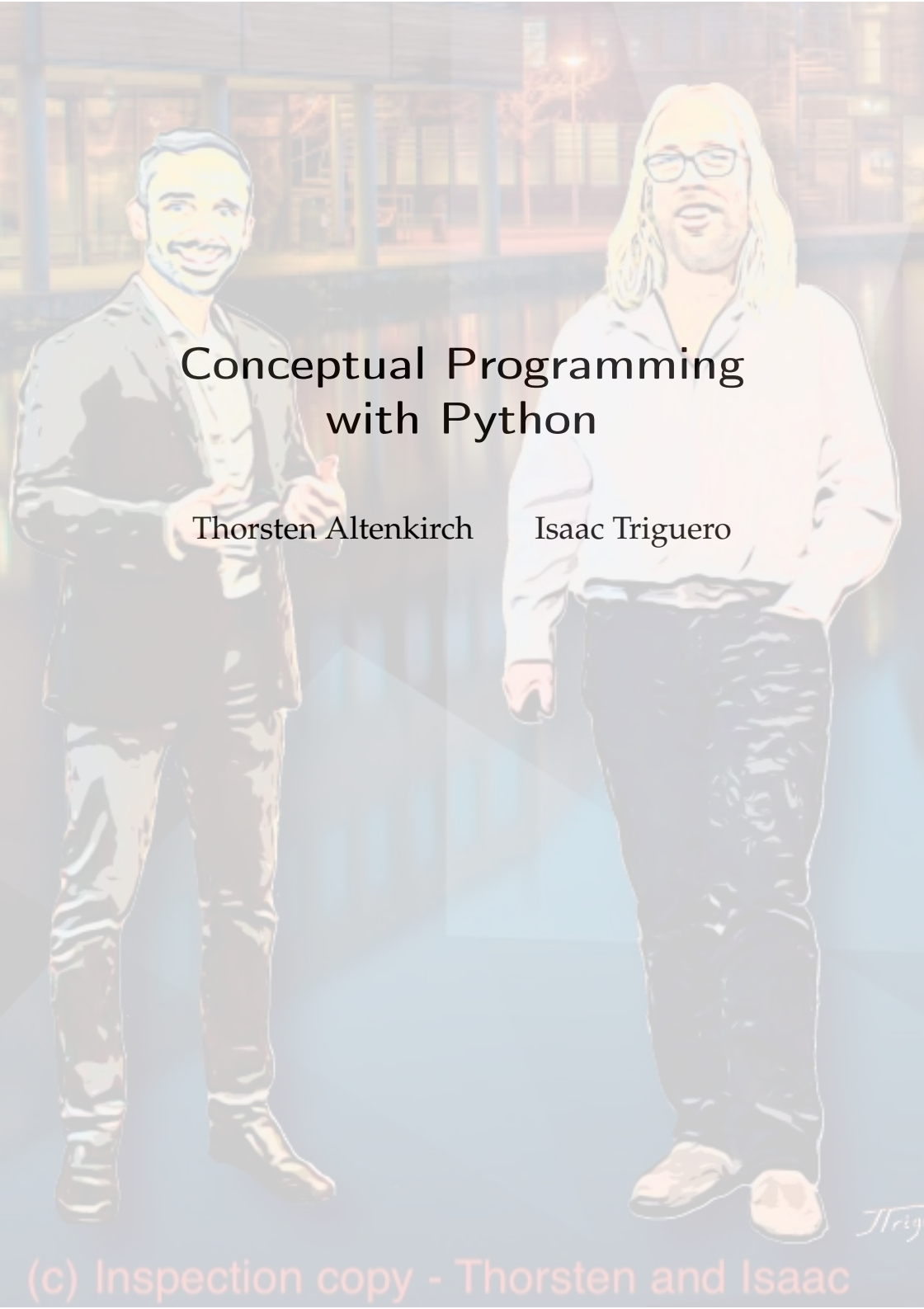


Conceptual Programming with Python

Thorsten Altenkirch Isaac Triguero

ISBN 978-0-244-82276-7

Contents

1 Introduction

This book is based on a course for Master's Students at the School of Computer Science of the University of Nottingham. The course and the book are intended for students with little or no background in programming coming from different backgrounds educationally as well as culturally.

It is not mainly a Python course, but we use Python as a vehicle to teach basic programming concepts. Hence, the words *conceptual programming* in the title. The concepts covered are:

- data structures and a visual understanding of their representation on a computer,
- control structures in imperative programming,
- the Halting problem showing the limits of computability,
- the use of recursion to design algorithms,
- backtracking to solve hard problems,
- the basics of object oriented programming,
- the use of trees to represent structured data,
- concepts of functional programming,
- how to write an interpreter,
- basic software engineering via a game development project,
- the basics of data science.

This includes some material which we do not cover in our lectures due to lack of time.

1.1 Why Python?

Python is a modern language which is named after *Monty Python* the British comedians, not the snake. We are using Python for the course and this book for the following reasons:

- Python has a very simple syntax with very little overhead. It uses layout to represent structure which is very natural and easy to read.
- Python uses dynamic typing, this makes it easy to learn because you don't have to get your head around a static type system, but see below.
- Python allows you to use concepts from a variety of programming paradigms, including object oriented programming and functional programming.
- There are a number of tools which make Python easy to use, like *jupyter notebooks* which we are using.
- Python features a *toplevel* like many functional languages, which makes it easy to interactively explore the language.
- Python is very popular, which results in a number of libraries (APIs) available in Python, which often makes it the language of choice in practice.

But it is hardly perfect. Here are some issues we have experienced with Python and which may make it a good idea to also look for other languages:

- The fact that Python doesn't use static typing means that many errors which would be flagged by other languages go undetected and may cause hidden errors in the software. These also means that interfaces are not clearly defined making the development of large systems harder.
- Python makes it often hard to use modern concepts, like recursion, because you have to pay an unnecessary performance penalty.
- Python also lacks certain features, like a pattern matching and algebraic data types, making the representation often unnecessarily clumsy.
- The lack of types leads to certain design errors in Python, for example the decision to avoid characters and represent them as strings.

However, weighing the reasons in favour and against we found that Python is the best choice for a course for beginners.

Our emphasis on concepts is important: you should be able to use them in any language you use for developing software.

1.2 How to use Python?

First of all you need to install Python, however it may already be installed since it is becoming a standard language. We are using the current version of Python which is Python 3.x ($x \geq 4$). We are using the Anaconda distribution which comes with a number of useful tools, but any other implementations should work fine too. However, we recommend using a version of Python that supports the use of a toplevel, which is not the case for some development environments.

Using anaconda there are a number of ways to interact with Python:

- using the toplevel. This isn't specific to Anaconda, you just type `python` on your terminal and talk to the Python interpreter.
- using jupyter notebooks. That is a nice way to combine software development, exploration and documentation. Indeed, this book was written this way.
- using an integrated development environments (IDE) like *spyder* which comes with anaconda, or *idle* which is part of the standard distribution.

1.3 Where to find more information?

There is so much material available on the internet now, that the students often get overwhelmed and confused. We have written this book to try to provide one consistent source of information for a course and suggest not to listen to too many chefs at the same time. That may spoil the soup.

On the other hand, we have tried to keep the material light, and we do not provide a complete reference manual to everything you may need. We recommend the following sources for additional information:

- The Python Tutorial[1] is an excellent source of information.
- If you really want to know the details of some aspect of the language, check out the Python Reference manual[2], but be warned this is like reading a law book to find out about a legal problem.
- Often the standard library is more important than the language itself: check out the Library reference manual[3].
- For specific projects you need to consult the API documentation. For example for working with *pygame* you should consult the Pygame docs[4].
- Finally, no programmer can survive without Stackoverflow[5] any more, a rich repository of questions and answers and you can join and ask your own questions (and add your own answers). But be warned you can get very confused and spend a lot of time looking through stackoverflow conversations which in the end turn out to be irrelevant for your issue.

1.4 Overview over the book

We start with an exploration of Python from the top-level (Chapter 2), which covers some basic concepts like data types, coercions, functions and so on. We also introduce a graphical view of data structures in Python. Next we look at imperative programming (Chapter 3) which is the traditional way of programming present in Fortran or C. We introduce basic control structures like if-then-else and loops. We also discuss Turing's famous *Halting problem*. After this, we introduce one of the most powerful spells the young software wizard should master: recursion (Chapter 4), that is a function that calls itself. We are also using this to implement a sudoku solver via back-

[1] https://docs.python.org/3.6/tutorial/index.html
[2] https://docs.python.org/3.6/reference/
[3] https://docs.python.org/3.6/library/index.html
[4] http://www.pygame.org/docs/
[5] https://stackoverflow.com

tracking. Obviously, we cover Object Oriented Programming (Chapter 5) which is now a standard approach to program development. We also explain the use of *trees* to represent expressions and knowledge bases. An alternative to Object Oriented Programming is Functional Programming (Chapter 6), which is close to a mathematical understanding of programming. We also cover infinite data structures and how to write a Python interpreter in a functional style. Now you need to develop a bigger program, ideally in a group, and we suggest writing a game because it is fun and it is easy to understand what the goal is, hence we introduce the pygame library (Chapter 7). Finally, we give an introduction to Data Science which underlies the modern approach to Machine Learning (Chapter 8).

We present some *challenges* during the text, which you should try to solve yourself, but our solution is provided at the end of the chapter. Each chapter finishes with a quiz and exercises. The quiz can be easily done by using the Python interpreter but the point is to see whether you understand the language well enough to execute programs in your head. Indeed, the ability to run programs in your head is essential if you want to be able to write programs. The exercises are of different degrees of difficulty.

1.5 About the authors

Thorsten Altenkirch (also known as `Der Chef`) and Isaac Triguero (also known as `El Jefe`) are with the School of Computer Science of the University of Nottingham.

Thorsten is from Berlin, Germany and has grown up on the western side of the wall. Indeed it is a little known fact that the wall was only built for him and it was taken down 6 weeks after he left to start his PhD in Edinburgh, Scotland. Having worked as a programmer in Berlin for various companies, Thorsten got sucked into more theoretical realms doing a PhD on Type Theory which is a synthesis of logical reasoning and functional programming. Following this ambition he has been working in Gothenburg, Sweden and even in (for a Prussian

from Berlin) more exotic places like Munich in catholic Bavaria. Eventually at the turn of the Millennium, Thorsten joined the School for Computer Science at the University of Nottingham where he founded the *Functional Programming Laboratory* together with his colleague Graham Hutton (check out Graham's Haskell book!). He ended up teaching Python after a sabbatical at the Institute for Advanced Study in Princeton which left him no other choice after his return. However, he has been enjoying teaching this course, especially since he was joined by his colleague Isaac.

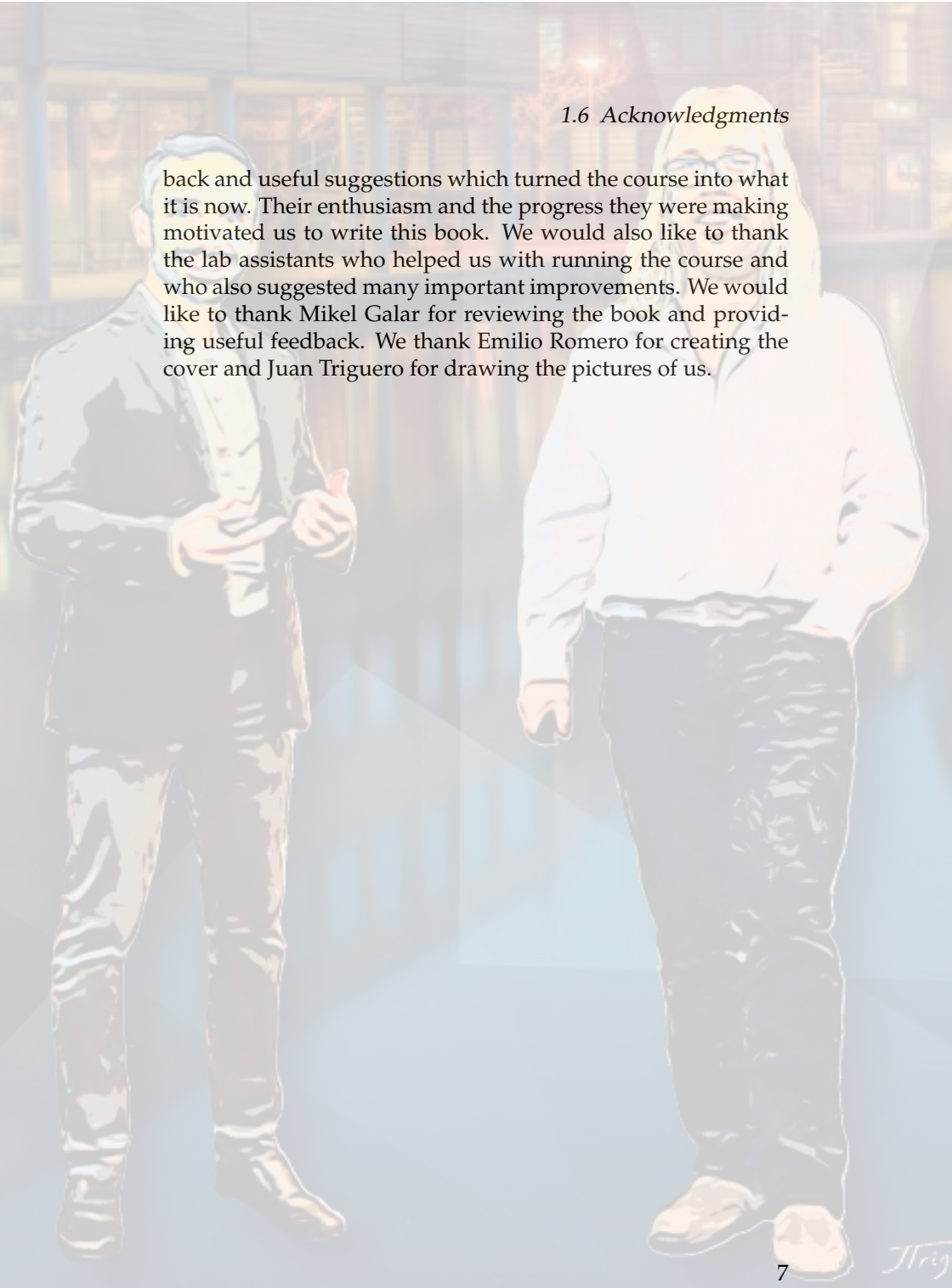
Isaac was born and bred in a small town, called Atarfe, in the region of the magnificent Granada, Spain, where the emblematic Arabic palace and fortress 'Alhambra' sits. Unfortunately, he can't tell the Alhambra was built for him -- and luckily it was not demolished after his departure --, but he can undoubtedly say that it is way more beautiful than that Wall from Germany. Isaac studied his MSc and PhD degrees in Computer Science at the University of Granada, where he was drawn into all the buzz words of the moment (Data science and Big data) before departing to Ghent where he worked (and mostly ate waffles on a daily basis) as a postdoctoral researcher. A few years ago, Isaac joined the School of Computer Science at the University of Nottingham, and he was soon severely punished and trapped into teaching this programming course with the inimitable Thorsten. As a "junior" professor, working with the old and wise `Der Chef` and his bold teaching style has been a real challenge for him, but it turned to be a very rewarding experience, in which they ended up teaching each other quite a few things.

Thorsten and Isaac have mostly written this book together, but they led different chapters; you are challenged to guess who wrote what...

1.6 Acknowledgments

We would like to thank the students who have attended our course in the previous years and have provided a lot of feed-

back and useful suggestions which turned the course into what it is now. Their enthusiasm and the progress they were making motivated us to write this book. We would also like to thank the lab assistants who helped us with running the course and who also suggested many important improvements. We would like to thank Mikel Galar for reviewing the book and providing useful feedback. We thank Emilio Romero for creating the cover and Juan Triguero for drawing the pictures of us.



2 Python from the top-level

We can explore Python interactively by using the top level. This means we can type in Python code and Python will answer directly. Python has inherited this concept from functional programming languages like LISP or Haskell.

2.1 Basic types and operations

We can use Python as a calculator.

3+5

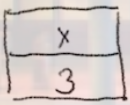
Here we exploit the top-level built into *jupyter* which allows us to interactively evaluate Python programs. After `In:` you see what we have written, and after `Out:` you see Python's response.

We also have variables.

```
In : x=3

In : x+5

Out: 8
```

Did you notice? There is no `Out:` after the `x=3`? That is because Python doesn't produce any output after an assignment.

As usual in programming the '=' sign has a different meaning than in Mathematics. Here it means that we want to store something in a variable, which I draw as a shoebox which has a label on it.



Unlike in many other programming languages we don't have to declare variables before using them. But they have to be initialised before we use them - otherwise we get an error.

```
In : x+y

   ---------------------------------------------
   NameError Traceback (most recent call last)
   <ipython-input-4-259706549f3d> in <module>() ---> 1x+y
   NameError:  name 'y' is not defined
```

Instead of `Out:` we see an error message. Error messages can be confusing: first we see a *Traceback* telling us where the error occurred during execution (pretty obvious in this case) and then the type of error (here a *NameError*) and the error message.

We also have floating point numbers:

```
In : 3/4

Out: 0.75
```

And there are strings for text:

```
In : "Thor"

Out: 'Thor'
```

One can use either single `'..'` or double `".."` quotes but the toplevel prefers to use `'..'` it seems. Sure, we can store strings in variables too.

```
In : me='Thor'

In : me

Out: 'Thor'
```

Data objects have types. We can use the function `type` to find out what type they are.

```
In : type(3+4)

Out: int
```

`int` stands for integer.

```
In : type(3/4)

Out: float
```

`float` is a floating point number.

```
In : type(me)

Out: str
```

and `str` is a string.

Note that it is not the variable `me` that has the type `str` but the object which is stored in it. We can also reuse `me` and store an integer.

```
In : me=42

In : type(me)

Out: int
```

This is called *dynamic typing*, because all the types in a program are determined when you run the code. The alternative is *static typing* where all the variables and operations have a fixed type which is known before you run your program. Static typing has the advantage that it avoids many errors but for beginners, dynamic typing is easier to understand.

2.1.1 Coercions

The operation + also works for strings. It means concatenation.

```
In : me+'sten'
```

```
---------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-15-6d2bddc8ae7e> in <module>() ---> 1me+'sten'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Oops, I forgot that I reused `me`. You see how error prone dynamic typing is - because this sort of error may occur when you run your program! Let's fix that.

```
In : me='Thor'

In : me+'sten'

Out: 'Thorsten'
```

Ok. I inadvertently also demonstrated that we cannot mix numbers and strings when using +.

```
In : me+3

---------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-18-391876a55bc2> in <module>() ---> 1me+3
TypeError: Can't convert 'int' object to str implicitly
```

We can convert data objects between different data types. For example we can convert a number to a string:

```
In : str(3)

Out: '3'

In : me+str(3)

Out: 'Thor3'
```

We can also go the other way if we have a string that contains a number.

```
In : int('5')

Out: 5
```

These operations are called *coercions*. Python usually requires explicit coercions, while many other languages use implicit coercions (that is, they happen automatically). I prefer the Python approach while it is more verbose it is less error prone.

2.1.2 Bool

Another useful type is `bool` the type if booleans, or truth values.

```
In : type(True)

Out: bool

In : type(False)

Out: bool
```

We can also use logical operations on booleans

```
In : True and False

Out: False

In : True or False

Out: True

In : not True

Out: False
```

We can also use `&` for `and` and `|` for 'or'. But be wary they behave slightly different.

```
In : True & False

Out: False

In : True | False

Out: True
```

Some operations return booleans, for example the test for equality ==. You see this is very different from =.

```
In : x == 3

Out: True

In : x == "3"
```

```
Out: False
```

And we can combine the test with logical operations.

```
In : (x == 3) or (x == "3")

Out: True

In : (x == 3) and (str(x) == "3")

Out: True

In : type(x) == int

Out: True
```

You ask what is the difference between `and` and `&` for example? Ok here it is:

```
In : x=0

In : not (x==0) and 1/x==0

Out: False

In : not(x==0) & 1/x==0

   ---------------------------------------------------
   ZeroDivisionError Traceback (most recent call last)
   <ipython-input-25-9bf9e76be849> in <module>() ---> 1not(x==0) &
1/x==0
   ZeroDivisionError:  division by zero
```

In the first case Python decided that it didn't need to evaluate the 2nd part because `false and` anything is `false`. In the 2nd version it did evaluate the 2nd part which led to an error because we tried to divide by `0`.

By the way, a shorthand for `not (x==0)` is `x!=0`.

2.1.3 Functions

The basic idea of a function is that it is a box where you can put something in and you get something out. We have functions in Python.

Let's define a function that adds 3 to its input and returns the result.

```
In : def f(x) :
         return x+3

In : f(2)

Out: 5
```

We are using a *parameter* in the definition of the function, I called it `x`. The *argument* of the function, e.g. `2` is assigned to the parameter `x` before the function is run. That is, there is a hidden assignment `x = 2` which happens before the body of the function is executed. The parameter `x` only exists while the function is executed, it is not visible from outside.

Did you notice the `:` and the indentation? The `:` indicates that we start a new *block* of code which has to be indented. Later we will see that we can nest blocks which means we have to indent further. The combination of `:` and indentation replaces the use of `{` and `}` you see in many C-like languages. We will say more about this in the next section.

We use the keyword `return` to indicate what the function *returns*. We will soon see that there are also Python functions that don't return anything.

Here is another function that doubles its input:

```
In : def g(x) :
         return x+x

In : g(2)

Out: 4
```

We can combine both functions in calculations.

```
In : f(g(2))
```

```
Out: 7

In : g(f(2))

Out: 10
```

Did you notice when we say `f(g(2))` we run **first** `g` and **then** `f` even though we write first `f` and then `g`. This is the curse of Mathematics. We should really write something like $((2)g)f$ but it is too late to change that.

There is a type of functions:

```
In : type(f)

Out: function
```

Since Python doesn't have static types it doesn't have more specific type for functions either. A function can take any input and produce some output.

We can have functions working on strings too.

```
In : def talk(who,what) :
         return who+" is "+what

In : talk("Thor","stupid")

Out: 'Thor is stupid'
```

At the same time `talk` is also an example for a function that takes more than one paraeter. No surprises here, I hope. We can use any number of parameters.

Let's define a function `isEven` which returns `True` if the input is an even integer and `False` otherwise.

We need to use the modulo operation % which calculates the remainder of a division. E.g.

```
In : 14%4

Out: 2
```

Because 14 divided by 4 is 3 remainder 2.

A number is even if the remainder from the division by 2 is 0.

```
In : 3%2

Out: 1

In : 4%2

Out: 0
```

Hence, we can define `isEven` as follows.

```
In : def isEven(n) :
         return n%2 == 0

In : isEven(5)

Out: False

In : isEven(8)

Out: True
```

Does this function satisfy our specification?

```
In : isEven("Thor")

  ---------------------------------------------------
  TypeError Traceback (most recent call last)
  <ipython-input-55-b02fb287e1a6> in <module>() --->
lisEven("Thor")
  <ipython-input-52-77d3f3007053> in isEven(n)  1 def isEven(n) :
---> 2return n%2 == 0
  TypeError:  not all arguments converted during string
formatting
```

No :-(. We didn't say that it only works for integers but that it should return `True` if the input is an even integer and `False` otherwise.

Can we fix it?

```
In : def isEven(n) :
         return type(n) == int and n%2 == 0

In : isEven("Thor")

Out: False
```

```
In : isEven(3)

Out: False

In : isEven(4)

Out: True
```

What would happen if we had used `&` here instead of `and`?

Can you figure it out in your head without actually running the code? This is an important skill if you want to become a good programmer!

Usually you won't define functions on the top-level but keep them in a file and edit them there. However, for the purposes of this book we will continue to show functions in the notebook.

2.2 Data structures

We have seen data types like integers, floats and booleans. A data structure is a data type which can be used to store other data in it. Actually the first example are strings which we have already seen.

2.2.1 Strings

We have already seen strings and the operation + to concatenate strings. Now, we look at some operations to access strings.

```
In : s = "Thorsten"

In : s[3]

Out: 'r'
```

`s[n]` returns the n+1th character - this is called indexing. Note that we are counting from 0 as we always do in computer science.

```
In : s[0]

Out: 'T'
```

There is no special type for characters but `s[n]` returns a string (of length 1).

```
In : type(s[3])

Out: str
```

Apropos length, there is a built-in function `len` that works on strings.

```
In : len(s)

Out: 8
```

Getting the last character of a string (which has index `len(s)-1` since we start counting at `0`)

```
In : s[len(s)-1]

Out: 'n'
```

There is a shortcut for this:

```
In : s[-1]

Out: 'n'

In : s[-2]

Out: 'e'
```

This is quite particular to Python - this trick won't work in most other languages.

Instead of just one character we can also extract a part of a string. This is called *slicing*.

```
In : s[2:5]

Out: 'ors'
```

Note that this starts at index 2 (i.e. the 3rd character) and ends at index 4 (i.e the 5th character); that is one before 5.

What happens if the 2nd index is before the first?

```
In : s[5:2]

Out: ''
```

If we leave out the first index we start at the beginning:

```
In : s[:5]

Out: 'Thors'
```

If we leave out the last index we go until the end:

```
In : s[2:]

Out: 'orsten'
```

And if we leave out both? Exactly!

```
In : s[:]

Out: 'Thorsten'
```

This seems pretty useless at the moment but we will see...

Now a little challenge, construct a new string form `s` with the first and last characters swapped.

```
In : s[-1]+s[1:-1]+s[0]

Out: 'nhorsteT'
```

Has anything happened to `s` ?

```
In : s

Out: 'Thorsten'
```

Obviously not. Evaluating expressions doesn't change variables. In this respect, strings behave like numbers.

Can we turn this into a function?

```
In : def swap(x) :
         return x[-1]+x[1:-1]+x[0]
```

Let's test it.

```
In : swap(s)

Out: 'nhorsteT'

In : swap("Python")

Out: 'nythoP'
```

What happens if we use *swap* twice?

```
In : swap(swap(s))

Out: 'Thorsten'
```

swap doesn't work for the empty string:

```
In : swap("")

  ---------------------------------------------------
  IndexError Traceback (most recent call last)
  <ipython-input-24-5b5714bf0d6e> in <module>() ---> 1swap("")
  <ipython-input-20-46b07c57f25c> in swap(x)  1 def swap(x) :
---> 2return x[-1]+x[1:-1]+x[0]
  IndexError:  string index out of range
```

We get an error if we access a string beyond its length. In the case of the empty string the index `0` is already out of range.

But what happens to a string containing only one character?

```
In : swap("a")

Out: 'aa'
```

Question: Why did this happen? Can you figure it out?

2.2.2 Lists

Lists are like strings but they are sequences of anything while strings are sequences of characters. For example a sequence of numbers.

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

In : type(ns)
```

```
Out: list
```

The items in a list can have different types, and in particular a list can contain lists again.

```
In : mixed = [1,"abc",[2,3,4]]
```

We can also coerce from and to lists, e.g.

```
In : list(s)

Out: ['T', 'h', 'o', 'r', 's', 't', 'e', 'n']

In : str(ns)

Out: '[1, 2, 3]'
```

Coercing into a string always produces the string that you see when you print the object.

All the operations we have seen on strings also work on lists.

```
In : ns+ns

Out: [1, 2, 3, 1, 2, 3]

In : ns+mixed

Out: [1, 2, 3, 1, 'abc', [2, 3, 4]]

In : ns[1]

Out: 2

In : mixed[2]

Out: [2, 3, 4]

In : mixed[1:3]

Out: ['abc', [2, 3, 4]]

In : mixed[2][2]

Out: 4
```

The last example shows that we can repeat indexing operations (e.g. mixed[2] provides a list, so, mixed[2][2] is accessing the 3rd component of the list provided in mixed[2]). We can actually apply indexing to something else than a variable. Indeed, we can even write:

```
In : "Thor"[1]

Out: 'h'
```

This is useful when we want to represent a 2-dimensional structure, e.g. a matrix.

```
In : mat = [[1,2,3],[4,5,6],[7,8,9]]
```

Or maybe a better layout is:

```
In : mat =
     [[1,2,3],
      [4,5,6],
      [7,8,9]]

In : mat[1][2]

Out: 6
```

Do you remember the function `swap` we have defined earlier? Does it work for lists as well?

```
In : swap(ns)

   ---------------------------------------------------
   TypeError Traceback (most recent call last)
   <ipython-input-37-8cb49238f8fa> in <module>() ---> 1swap(ns)
   <ipython-input-20-46b07c57f25c> in swap(x)  1 def swap(x) :
---> 2return x[-1]+x[1:-1]+x[0]
   TypeError:  unsupported operand type(s) for +:  'int' and
'list'
```

The problem is that the component of a list is usually not a list. In this case l[-1] is a number and we cannot + numbers and lists.

We can define a special version of swap for lists. We can turn an element into a list by putting [..] around it. E.g.

```
In : ns[-1]

Out: 3

In : [ns[-1]]

Out: [3]
```

We can use this to swap a list:

```
In : [ns[-1]]+ns[1:-1]+[ns[0]]

Out: [3, 2, 1]
```

Let's turn this into a function:

```
In : def swapl(xs) :
         return [xs[-1]]+xs[1:-1]+[xs[0]]

In : swapl(ns)

Out: [3, 2, 1]

In : swapl(list(s))

Out: ['n', 'h', 'o', 'r', 's', 't', 'e', 'T']
```

But this operation doesn't work for strings.

```
In : swapl(s)

   ---------------------------------------------------
   TypeError Traceback (most recent call last)
   <ipython-input-48-8daeb6f87815> in <module>() ---> 1swapl(s)
   <ipython-input-41-ebb51c35ddb6> in swapl(xs)  1 def swapl(xs) :
---> 2return [xs[-1]]+xs[1:-1]+[xs[0]]
   TypeError:  can only concatenate list (not "str") to list
```

Challenge #1: Can we fix this? Can we define one function that works both for strings and for lists? (Section 2.5)

2.2.3 Modifying lists

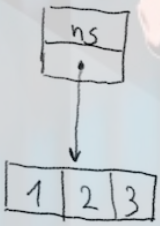
We can change lists. For example given my favourite list

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

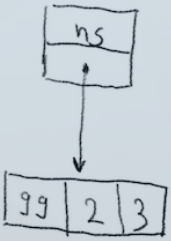
Let's say I want to change the 1st element (that is index `1`) to `99`. Here we go:

```
In : ns[0] = 99

In : ns

Out: [99, 2, 3]
```

That was easy. We can view a list as consisting of little boxes that can be changed individually just like the labelled boxes which correspond to variables. That is before the assignment we would draw the following image:



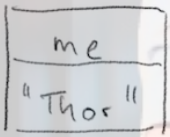
and after it looks like this:



Before we have seen that the same operations worked for strings and for lists. If you now think that the update operation will also work for strings you will be disappointed:

```
In : me="Thor"

In : me[1]="x"

---------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-7-dd2ce9adf30b> in <module>() ---> 1me[1]="x"
TypeError: 'str' object does not support item assignment
```

To make this visible I write a string directly into the variable box (the same as for numbers):



We say that strings are *immutable*, unlike lists which are *mutable*.

You may wonder why, especially since other programming languages do allow you to update strings. One reason is that knowing that you don't update strings can make the code more efficient because you can reuse a string as many times as you like.

However, we can always turn a string into a list first:

```
In : meList = list(me)

In : meList

Out: ['T', 'h', 'o', 'r']

In : meList[1] = "x"

In : meList

Out: ['T', 'x', 'o', 'r']
```

Ok let's do the swap operation we have seen previously but this time we are going to modify a list.

First of all we restore the list (this is the problem with *destructive* operations).

```
In : meList = list(me)
```

Ok we are going to swap the first and the last element of the list, i.e. the items at index `0` and `-1`.

```
In : meList[0]=meList[-1]

In : meList[-1]=meList[0]
```

Let's see whether this has worked.

```
In : meList

Out: ['r', 'h', 'o', 'r']
```

Oops! This is not what I had in mind.

What has happened? Let's see, first restore the list (again).

```
In : meList = list(me)
```

Then do the first step.

```
In : meList[0]=meList[-1]

In : meList

Out: ['r', 'h', 'o', 'r']
```

Ok, now it is clear. When we did the 1st update we lost the original first character. Hence we have to save it before we change it.

Ok, let's start again.

```
In : meList = list(me)
```

We first store the first character:

```
In : helper = meList[0]
```

Then we copy the last character to the front.

```
In : meList[0] = meList[-1]

In : meList

Out: ['r', 'h', 'o', 'r']
```

And now we set the last character to the saved first character.

```
In : meList[-1] = helper
```

And voila!

```
In : meList

Out: ['r', 'h', 'o', 'T']
```

We can put this into a function. I call it `swapx` where the `x` stands for change.

```
In : def swapx(lst) :
         helper = lst[0]
         lst[0] = lst[-1]
         lst[-1] = helper
```

`swapx` is our first example of a function that doesn't return anything but it just does something.

We can use this to restore meList.

```
In : meList

Out: ['r', 'h', 'o', 'T']

In : swapx(meList)

In : meList

Out: ['T', 'h', 'o', 'r']
```

What is the difference between the function `swapl` and `swapx`? Both work on lists.

`swapl` returns a list but doesn't change the list.

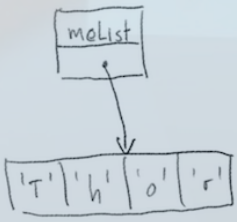
```
In : meList
```

```
Out: ['T', 'h', 'o', 'r']

In : swapl(meList)

Out: ['r', 'h', 'o', 'T']

In : meList

Out: ['T', 'h', 'o', 'r']
```

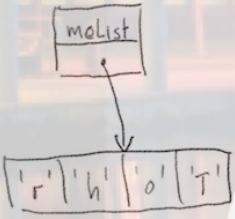
While `swapx` doesn't return anything but changes the data structure.

```
In : meList

Out: ['T', 'h', 'o', 'r']

In : swapx(meList)

In : meList

Out: ['r', 'h', 'o', 'T']
```

While swapx is called a *function* in Python, it isn't really a mathematical function. It changes the memory of the computer, it modifies boxes. E.g. we can draw the memory before running `swapx`



and after:



Ok, how do we swap twice? (Admittedly not a very useful operation).

For `swapl` it works like this.

```
In : meList = list(me)

In : swapl(swapl(meList))

Out: ['T', 'h', 'o', 'r']
```

And for `swapx`?

```
In : swapx(swapx(meList))

  ------------------------------------------------
  TypeError Traceback (most recent call last)
  <ipython-input-46-aa6bb992929e> in <module>() --->
1swapx(swapx(meList))
  <ipython-input-30-c5848f168542> in swapx(lst)  1 def swapx(lst)
:  ---> 2helper = lst[0]  3 lst[0] = lst[-1]  4 lst[-1] = helper
  TypeError:  'NoneType' object is not subscriptable
```

This is a strange error. What has happened?

As I said `swapx` doesn't return anything. Actually it returns `nothing`. And this is fed into the outer call of `swapx` which doesn't know what to do with it.

To do `swapx` twice we just have to run it twice. Like this:

```
In : meList = list(me)

In : meList

Out: ['T', 'h', 'o', 'r']
```

```
In : swapx(meList)
     swapx(meList)

In : meList

Out: ['T', 'h', 'o', 'r']
```

In Python there is actually a way to define swapx without using the helper variable. Python allows a parallel assignment, that is we can update both boxes in parallel:

```
In : def swapx(lst) :
         lst[0],lst[-1] = lst[-1],lst[0]
```

2.3 Mysteries of Python

Can we write a `swapx` function that works for strings?

Clearly, we cannot use the assignment operation to a component of a string. But on the top-level we can change a variable that contains a string. E.g.

```
In : me = "Thor"

In : me = swap(me)

In : me

Out: 'rhoT'
```

So what stops me from turning this into a function?

```
In : def swapsx(s) :
         s = swap(s)
```

Ok, let's test this.

```
In : me = "Thor"

In : swapsx(me)

In : me

Out: 'Thor'
```

This was not what we expected! No change!

What has happened? The variable `s` is a parameter, it is a box that only exists while we execute the function `swapsx`. When we call `swapsx(me)` we copy the *content* of `me` into `s`. So indeed at the end of the execution of `swapsx` the box `s` contains the swapped version of the string but nothing has happened to `me`.

There is no operation in Python (unlike e.g. in C) which creates a reference to a top-level variable - hence it is not possible to write a function that updates an arbitrary top-level variable. However, it is possible to update a specific top-level variable.

```
In : def swapme() :
         global me
         me = swap(me)

In : me

Out: 'Thor'

In : swapme()

In : me

Out: 'rhoT'
```

The declaration `global me` tells Python that `me` is a *global* variable and not a local one.

We have already seen that we can update lists in a function, which is because they are a data structure that contains boxes and we update those boxes and not the content of the variable box on the top-level.

But can we use the `swapl` function instead of doing the assignments? Clearly, if we just do the same as for strings it won't work. That is the following attempt:

```
In : def swaplx(lst) :
         lst = swapl(lst)
```

It doesn't work. But there is a clever way to make this work. It turns out that in Python we can also replace slices of lists. That is for example:

```
In : meList = list("Thor")

In : meList

Out: ['T', 'h', 'o', 'r']

In : meList[1:]="im"

In : meList

Out: ['T', 'i', 'm']
```

And hence using the apparently useless `lst[:]` we can change the whole list.

```
In : def swaplx(lst) :
         lst[:] = swapl(lst)

In : meList = list("Thor")

In : swaplx(meList)

In : meList

Out: ['r', 'h', 'o', 'T']
```

2.4 Summary

We can evaluate expressions or assign values to variables using `x = e` where `e` is some expression.

Basic types in Python :

- Integers (`int`)
- Floating point numbers (`float`)
- Strings (`str`)
- Truthvalues (`bool`)
- Functions (`function`)
- Lists (`list`)

Use `type(..)` to find out the type of an object. Use the name of a type as a function to coerce a value to that type, if possible.

Integers and Floats

We have the usual arithmetic operations like `+`, `-`, `*` and `**` (exponentation). Note that `/` on integers produces a float while `//` will produce an integer while `%` computes the remainder.

Strings and Lists

String constants can be written either `"..."` or `'...'`. Lists are written as `[ a1 , a2 , .. ]`.

`+` is concatenation. You can access the nth element of a string or a list using `x[n]`, we start counting from `0`. Using negative numbers counts from the end where `-1` is the last item. You can slice using `x[m:n]`; this produces the substring/list starting at `m` and ending at `n-1`. If you leave out the first index the default is 0 and if you leave out the last the default is `-1`. `len(x)` computes the length of a string or a list.

You can modify lists by using index or slice expressions on the left hand side of an assignment. Strings cannot be modified.

Truthvalues

Basic truthvalues are `True` and `False`. We can combine truthvalues using `and`, `or` and `not`. Alternatively we can use `&` and `|` which always evaluate all components.

Basic predicates return truthvalues such as == equality and <,<=,> and >= which compare numbers and other data types. `!=` is inequality.

Functions

Functions are defined starting with

```
def f (x1 , x2 , .. , xn) :
```

followed by an indented block of code. Here `xi` are the parameters of the function. The block may contain a statement `return e` which means that the function returns the value

e. Variables in functions are local unless they are explicitly marked as global using

```
global x1,x2,..
```

2.5 Solution to top-level challenge

The challenge was to create a program that would perform the functional swap operation both for lists and for strings. The trick is to use slicing instead of indexing because it works the same for both lists and strings.

```
In : def swap (x) :
        return x[-1:]+x[1:-1]+x[:1]

In : swap(list("Thor"))

Out: ['r', 'h', 'o', 'T']

In : swap("Thor")

Out: 'rhoT'
```

2.6 Quizzes

```
In : myString = [["1","a"],0,[""]]
```

what is the output of each of the following lines. Try to figure this out in your head and on paper without using Python first - then you can check you answers using Python.

1. `myString[0][0][0]`
2. `myString[myString[1]][0]+myString[2]`
3. `myString[0][1]+myString[2][0]`
4. `myString[0:1]+myString[1:2]`
5. `myString[int(myString[0][0])]`

2.7 Exercises

1. Define a function rotateR with one argument (a string) that returns the string rotated to the right. E.g. `rotateR("Thor")` should return `"rTho"`.

2. Define a function rotateL with one argument (a list) that returns the list rotated to the left. E.g. rotateL([1,2,3,4]) should return [2, 3, 4, 1].

3. Do both functions work for lists and strings? If not, how can you fix that?

4. Define a function rotateRx that gets a list as a parameter and changes that list by rotating it to the right. The function should return nothing. E.g. we assign `l = [1,2,3,4]` and then run `rotateRx(l)` (which returns nothing). If we now check `l` it returns `[4, 1, 2, 3]`.

5. Can you modify 4. so that it works for strings?

6. Write a function `rotateR2` that gets a string as an argument and returns the string rotated to the left twice, using only the function `rotateR` from the 1st exercise. E.g. `rotateR2("Thor")` should return `'orTh'`.

7. Write a function `rotateRx2` that changes its parameter (a list) by rotating it twice to the right. The function should only use rotateRx from part 4. E.g. we assign `l = [1,2,3,4]` and then run `rotateRx2(l)` (which returns nothing). If we now check `l` it returns `[3, 4, 1, 2]`.

8. Can you create a list `l` that prints as `[[1, 2, 3], [1, 2, 3]]` but when I rotate only the first part by `rotateRx(l[1])` and then print `l` it turns out that both parts have been rotated, i.e. I get

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