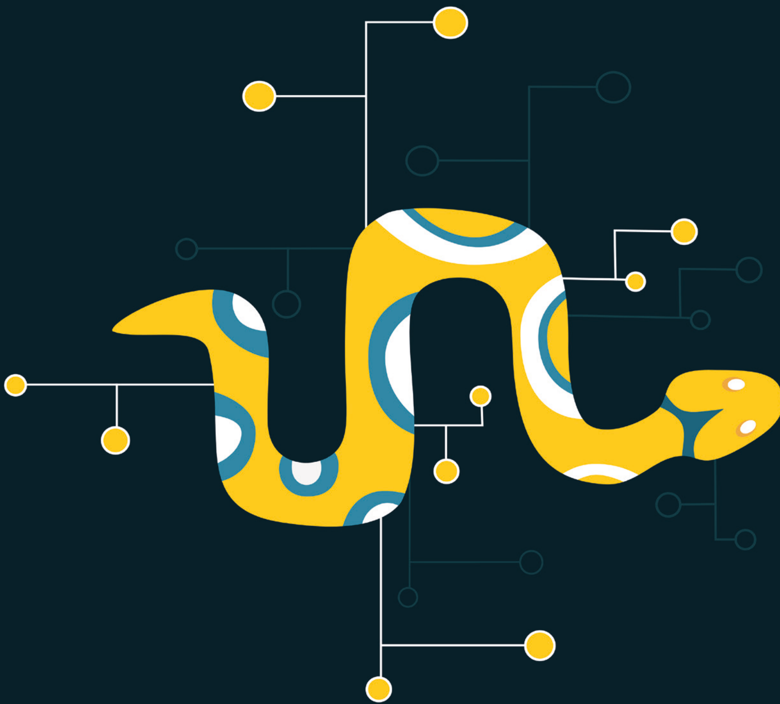
elektor books

Python 3 for Science and Engineering Applications

Learn to use Python productively in real-life scenarios at work and in everyday life



Felix Bittmann

elektor
design > share > sell

Python 3 for Science and Engineering Applications

Felix Bittmann

an Elektor Publication

LEARN DESIGN SHARE

This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
78 York Street
London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344
© Elektor International Media BV 2020
First published in the United Kingdom 2020

All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's written permission to reproduce any part of this publication should be addressed to the publishers. The publishers have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, and hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other cause.

British Library Cataloguing in Publication Data
Catalogue record for this book is available from the British Library

ISBN 978-3-89576-399-1

EBOOK 978-3-89576-400-4

EPUB 978-3-89576-401-1

Prepress production: DMC ¦ daverid.com
Printed in the Netherlands by Wilco

Elektor is part of EIM, the world's leading source of essential technical information and electronics products for pro engineers, electronics designers, and the companies seeking to engage them. Each day, our international team develops and delivers high-quality content - via a variety of media channels (e.g., magazines, video, digital media, and social media) in several languages - relating to electronics design and DIY electronics. **www.elektor.com**

LEARN DESIGN SHARE

Table of Contents

• Introduction . . . 6

Chapter 1 • Basics . . . 8

1.1 • Installation and Programming Environment . . . 8
1.2 • Basic Python . . . 8
1.3 • Principles of Good Programming . . . 18
1.4 • Problem-Solving Skills . . . 20

Chapter 2 • Working with Numbers . . . 22

2.1 • Fibonacci . . . 22
2.2 • Prime Numbers . . . 26
2.3 • Collatz . . . 29
2.4 • Pi . . . 30
2.5 • Countdown . . . 35
2.6 • Ulam Spiral . . . 42
2.7 • Total Chaos . . . 46
2.8 • Three Points . . . 55
2.9 • Close Together . . . 64
2.10 • Backtracking . . . 70
2.11 • Numerical Integration . . . 74

Chapter 3 • Statistics and Simulations . . . 79

3.1 • Speedtest . . . 79
3.2 • Pi (again) . . . 80
3.3 • Parallelisation . . . 84
3.4 • Random Walk . . . 88
3.5 • Game of Life . . . 92
3.6 • Modelling Populations . . . 96
3.7 • Quick Money . . . 102
3.8 • Many Circles . . . 106
3.9 • Pig . . . 114
3.10 • Bootstrapping . . . 124

Chapter 4 • Text Data and Strings . . . 132

4.1 • Dictionary . . . 132
4.2 • LPS . . . 135
4.3 • LCS . . . 137
4.4 • Encryption . . . 141
4.5 • Roman Numerals . . . 147
4.6 • Match Arithmetic . . . 149
4.7 • Superpalindromes . . . 154
4.8 • 2048 . . . 158
4.9 • The Next Steps . . . 164

• Introduction

Why Python?

Not without reason Python has become one of the most popular programming languages in the world. A user-friendly and intuitive syntax, a large and motivated community, paired with a multitude of modules and program libraries, which allow quick and efficient implementation of any project ideas inspire beginners and experts alike. Therefore Python is an ideal first step into programming but also recommended for veterans who would like to get a foothold in the realm of data sciences.

This book is written for readers who already have basic experience with Python, say after completing a first tutorial, and now want to learn how to apply Python productively and with a focus on applications in real-world settings. Therefore, this is not a classical textbook that processes all aspects of the language linearly but rather starts with very concrete tasks and puzzles that want to be solved. These are taken from a large number of different fields to emphasize that Python can be applied in many contexts. In each example, we will first look at the general ideas or tactics of how to solve the problem and when how these can be implemented with special Python tricks and tweaks.

Requirements

You should know about the basic usage and commands before starting with the present book. As long as you are informed about the most common data types (integers, floats, strings, lists, dictionaries), know how to write a simple function, and can deal with lists, you will be able to solve all problems posed in this book. If you want to have a quick refreshment of the most basic aspects of the language, I recommend the course offered by the University of Waterloo.[1]

Philosophy

The puzzles presented in this book are aimed at beginners with only a little experience with general topics of programming. If any mathematical techniques are necessary to solve a problem they will be introduced with the puzzle itself. The code shown in this book does not aspire to be the most elegant, shortest, or most performant solution but rather illustrates basic concepts of programming and how to think like a programmer. For most puzzles presented there exist highly specialized algorithms that can improve speed manifold but are often not obvious to beginners and require in many cases a lot of background information. To solve the problems you will not require any other tools, software, or packages than the native Python environment (*pure Python*). This being said, there exists a multitude of excellent Python packages that drastically increase the number of functions of Python (for example, *NumPy, SciPi* or *Pygame*, just to name few). However, these often come with extensive documentation and need tutorials to be comprehensible to the beginner. In general, the easier puzzles are placed at the beginning of a chapter to introduce new concepts and methods that are then assumed to be known in the following

1 https://cscircles.cemc.uwaterloo.ca/

puzzles. Therefore it might be a good idea to work on the problems following the order of the book. However, if you feel confident feel free to skip and play around. If there are any unknown commands or concepts, it is often the quickest way to hit up a search engine and look things up online since it only takes seconds and is the easiest way.

Acknowledgments

I am very thankful to all people who helped me with this book, especially Florian Scholze, Jannik Köster, and Kurt Bittmann. Simon Wolf checked the entire code meticulously and improved it beyond imagination. Without Tam Hanna, there would be no english version of this book: I am deeply grateful for this enthusiasm and mentorship. Furthermore, I want to thank the *Python Software Foundation* in general for donating this wonderful gift to the world. Finally, many thanks to all men and women who contribute to free open-source projects like *Wikipedia* and *Wikimedia Commons*, which allow me to include a large number of high-quality figures in this book.

All code available on: https://github.com/fbittmann/Pythonbook

Chapter 1 • Basics

1.1 • Installation and Programming Environment

Make sure you have installed the most up to date version of Python from python.org. To run the code presented in this book you need at least version 3.6. If you run Linux or Mac, the chances are Python is already pre-installed on your system. To test which version you are running, open a terminal (Linux or Mac) or the power shell (Windows). Then type python3 to start an interactive session. Then the current version will be displayed.

I recommend using Geany[1] as an IDE or editor. This smallish (16 MB) open-source application is perfect for beginners and advanced users and comes with many functions without being bulky or too complicated. Furthermore, a large number of themes, schemes, and plugins allow extending the basic functions easily. Geany is available for Linux, Windows, and Mac.

1.2 • Basic Python

The next few pages serve as a crash course and are recommended for all users who want to refresh their skills, so feel free to skip ahead if you want to. In contrast to most code shown in this book we will here refer to an interactive Python session, which is denoted by >>> to visualise the interactive character of the code. This means, type a line, hit enter and you will instantly see the result, which is different from writing a large script and then have it run as a whole. Output, if there is one, is then displayed in the following line without the >>>.

```
>>> a = 12
>>> b = 3.141
>>> c = "Tomato"
>>> d = [a, b, c]
>>> e = (1.734, 3.822)
>>> f = {3, 8, 99, -4}
>>> g = {"Hello": 5, "Nope": 4, "Ego": 3, "Rocket": 6}
```

Here, *a* is an integer, *b* a float, *c* a string, *d* a list, *e* a tuple, f a set and g a dictionary. As you see, declaring a variable only requires the equality sign. When working with mathematical expressions, make sure to remember *BEDMAS* (brackets, exponents, division, multiplication, addition, subtraction) since this helps you memorise the order in which operators are addressed. Note that longer blocks of code are split over multiple lines if necessary using "\" as an indicator for a line break. If you enter the code in your editor, do not type this sign as it is just a visual aid for the printed version.

Indices and Slices

For Python, lists are an all-purpose tool that can be utilised in most situations. Sets, tuples

1 Geany.org

and dicts add many more features and are often faster or more convenient, but Python loves lists. You can store any elements or data types in a list and of course also more and nested lists. You retrieve items from a list via their index. Remember, in Python (as in most other programming languages), the first item of a list always receives the index 0.

```
>>> a = [1, 2, 3]
>>> b = ["Hi", 1, "Red", -6.87, [1, 2, 3, ["Mouse"]], 95]
>>> a[0]
1
>>> b[2]
"Red"
>>> b[4][1]
2
>>> b[-1]
95
>>> len(b)
6
>>> len(b[4])
4
```

As you see, items in nested lists are retrieved by combining several indices directly. For example, if you want to retrieve the integer 2 from list b, first select the containing nested list (which has the index 4) and then the index of this sub-list (which is the index 1), so the final result is b[4][1]. Here, always use square brackets (this also holds for tuples and dicts). If you want to retrieve the last item of a list, regardless of the number of items contained, use negative indices. The last item always receives the index -1. The number of elements in a list is reported by using *len()*. If you want to cut a list in parts, we refer to this as slicing.

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> a[0:3]
[1, 2, 3]
>>> a[2:5]
[3, 4, 5]
>>> a[::2]
[1, 3, 5, 7]
>>> a[::-1]                 #Reverse a list
[7, 6, 5, 4, 3, 2, 1]
```

The slice-operator has three parts: the start, end, and step. The start is always included in the resulting list, the end is always excluded. If no step is explicitly set, 1 is implied. If start or end are omitted, Python uses the first or the last element. Also, note that lists and strings can be sliced in the same form.

```
>>> w = "Trebuchet"
>>> w[3]
"b"
>>> w[2::2]
"euht"
```

Dictionaries

Dictionaries or dicts are convenient when you want to build a very simple database for lookups. Here pairs of keys and values are created, which are not selecting by an index but by key. Let's have a simple example with dates of birth.

```
>>> dateofbirth = {"Dawkins": 1941, "Dostojewski": 1821, "Goethe": 1749}
>>> dateofbirth["Goethe"]
1749
>>> dateofbirth["Boyle"] = 1948
>>>   dateofbirth
{"Dawkins": 1941, "Dostojewski": 1821, "Goethe": 1749, "Boyle": 1948}
```

The first value (before the colon) is the key, the one after the value. To retrieve the value, just enter the key in brackets. Adding new items is done likewise. Note that keys must be immutable, so you can use integers, floats, strings, or tuples, but not lists. For values, any data type is fine. Dicts have the advantage over lists that a lookup is faster. A very common task is to loop over keys, values, or both and retrieve certain elements. Here you have several options to do this.

```
>>> for key in dateofbirth.keys():
>>>    key
Dawkins
Goethe
Dostojewski
Boyle

>>> for value in dateofbirth.values():
>>>    value
1941
1821
1749
1948

>>> for key, value in dateofbirth.items():
>>>    key, value
("Dawkins", 1941)
```

```
("Dostojewski", 1821)
("Goethe", 1749)
("Boyle", 1948)
```

The last scheme is especially useful since you retrieve both keys and values at the same time in a tuple and can work with them immediately. The order in which the elements will be retrieved from the dict was random until version 3.7, after that every dict comes with an inherent ordering, which might be useful for certain applications. Later we will see how we can sort dicts arbitrarily. As a side note: whenever we work in the interactive session as in the last example, it is optional to use the print-statement to generate output since just calling a variable or function will automatically produce a visual output in the console. However, if you want to use the same code in a file, always wrap these variables in *print()*, otherwise, it will not be on display.

Loops

Python knows several different ways of looping. Using *for*, you can directly loop over all elements of a given iterable or iterator, for example, a range, list, or tuple.[2] While-loops are useful when you do not know in advance how often a loop is executed and you want to exit dynamically. Let's have a look at three examples.

```
>>> for i in range(0, 10, 2):
>>>     i
0
2
4
6
8

>>> wordlist = ["This", "is", "fine"]
>>> for word in wordlist:
>>>     word
'This'
'is'
'fine'

>>> value = 0
>>> while value < 64:
>>>     value
>>>     value = 2 **  value
```

2 In Python an *iterable* is an object which can be iterated over, say a list or tuple. An *iterator* is a *generator* that saves its own internal state, which is useful when the same object is called again. Only iterators can be called using *next()*. Later we will see how this can be used for our benefit.

```
0
1
2
4
16
```

The first loop produces all even numbers from 0 to 10 (exclusively). As with slices, the first value is the start, the second the stop, and third the step. The variable i is the index and can be named arbitrarily. The second loop iterates over all elements of the given list. The last loop continues running until the exit condition is met. In this example, value has to be smaller than 64 so that the loop continues. If this condition is violated, the loop is not started anymore. Loops that never meet this exit condition will run forever and must be terminated by the user (infinite loop). Therefore, make sure the variable that controls the exit is manipulated somewhere inside the loop as only then an exit is possible.

If you want to exit a loop prematurely, use *break*. *Continue* is useful when you want to keep the loop running but skip over certain elements, possibly to improve performance or avoid obvious errors (like when you want to process integers but a string shows up in a list). With *continue*, Python will always skip to the start of the loop immediately, regardless of where the script executes at the moment within the loop. Use *pass* as a generic placeholder which does exactly nothing, as the name indicates. Let’s have a look at three examples.

```
>>> for number in range(1, 5):
>>>    print(number)
>>>    if number == 3:
>>>            break
>>>    print(number * 10)
>>> print("Outside loop now")
1
10
2
20
3
Outside loop now
```

As soon as *break* is reached, Python will leave the loop at once and continue with the code below. Any code within the loop below break will be skipped.

```
>>> for number in range(1, 5):
>>>    print(number)
>>>    if number == 3:
>>>            continue
>>>    print(number * 10)
>>> print("Outside loop now")
1
10
2
20
3
4
40
Outside loop now
```

When *continue* is reached, Python will go back to the start of the loop and continue with the next element of the iterable. The code inside the loop below continue is skipped.

```
>>> for number in range(1, 5):
>>>    print(number)
>>>    if number == 3:
>>>            pass
>>>    print(number * 10)
>>> print("Outside loop now")
1
10
2
20
3
30
4
40
Outside loop now
```

When *pass* is reached, nothing happens. The loop is not exited and Python continues to run any code below *pass* if there is any. *Pass* usually works as a placeholder.

Comprehensions

Comprehensions can be used as a very compact alternative for loops and might also improve performance. While we distinguish list, dict, set and generator comprehensions, their syntax is almost identical. Suppose you want to generate a list with all integers below 100 that are divisible by both 3 and 7. Using comprehensions we can solve this within one

line of code.

```
>>> [i for i in range(100) if i % 3 == 0 and i % 7 == 0]
[0, 21, 42, 63, 84]
>>> [i ** 2 for i in (1, 2, 3, 4, 5)]
[1, 4, 9, 16, 25]
```

The square brackets indicate that we want to create a list, i is the index which takes all values from 0 to 99. As you see we included a filter to sort out all integers that do not fit our condition. The second example illustrates how we can dynamically transform results before adding them to the list. *If...else* constructions are also allowed with a slightly different syntax (note the ordering of the elements).

```
>>> [1 if x > 5 else 0 for x in range(10)]
[0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
```

In this example, we receive a list that displays a 1 for any number that is larger than 5 and a 0 otherwise. If and else are now placed on the left side of the iterator since this is not a filter any more but the ternary operator. Sets and dicts can be created likewise, the only difference is the type of brackets.

```
>>> {i for i in range(10)}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {word: len(word) for word in ["We", "have", "fun"]}
{'We': 2, 'have': 4, 'fun': 3}
```

Be aware of the fact that comprehensions can become easily complex when nested comprehensions are included. In this example, we create a simple matrix, which is a list with sub-lists.

```
>>> [[i * j for i in range(4)] for j in range(4)]
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6], [0, 3, 6, 9]]
```

Python works from inside out, first creating a list that contains the products of i and j. After that, the four new lists are returned together in one superior list. As you see, this gets difficult to read and while comprehensions allow for very compact and sophisticated expressions, they can easily become a nuisance for colleagues (or yourself after returning to your code after a two-week break). Whenever loops are nested, special caution is advised

to generate benign and readable code.

Functions

Whenever you need to solve more complex tasks it is strongly advised to split up your code into functional parts and create several combined functions. This has many advantages: firstly, functions can be easily reused and even imported into other documents. Secondly, debugging functions is often easier than larger blocks of code since you can test each function separately. Summarised: divide and conquer!

In Python, functions can be defined with two expressions. The first one is *def()*. A function can include an arbitrary number of arguments, which can also be set as defaults.[3] Let's see this in action with a very simple calculator for addition.

```
>>> def adder(x, y):
>>>     return x + y
>>> adder(1, 1)
2
```

This function has two arguments, x, and y. These must always be specified by the user when calling the function. Using *return* we specify which value we want to receive back from the function. If no return is set by the programmer or if it is never reached, the function will then return *None*. In many cases, this is irrelevant, for example, when a function is used only to display something in the interactive session.

```
>>> def greetings(name):
>>>     print("Hello " + str(name) + "!")
>>> greetings("Python")
"Hello Python!"
```

Using defaults we can pre-specify certain arguments that can be overwritten by the user if desired.

```
>>> def exponentiate(x, y=2):
>>>     return x ** y
>>> exponentiate(3)
9
>>> exponentiate(2, 4)
16
```

3 In this book the terms *parameters* and *arguments* are used changeably regarding functions.

We can also create anonymous functions using *lambda*. These functions are usually very compact as they consist of only one expression and can be defined "on the fly".

```
>>> adder = lambda x, y: x + y
>>> adder(2, 2)
4
```

As you need to restrict the functionality to one expression, these are usually not applicable to more complex tasks. At this point, you should also be aware of the fact that certain expressions can be shortened to make code more compact.

```
x = x + 5 <=> x += 5
x = x - 5 <=> x -= 5
x = x * 5 <=> x *= 5
x = x / 5 <=> x /= 5
```

Internal Checks and Dealing with Exceptions

Writing software for end-users requires a lot of time and effort to make sure that inputs are sanitised and only certain data types are fed into special functions. For example, a calculator app should never have to deal with strings since only numbers are used for arithmetic. When writing code for a web application, make sure that an email address always contains exactly one at sign (@). In some cases, the receiving function will notice the problem and throw an exception or error message, which is usually a good thing since you will be alerted that something went wrong. Sometimes these issues go unnoticed and the first problem you will notice is way down the line, maybe after receiving a wrong result. Finding the bug then can be tedious and difficult so creating a few checkpoints is often a good idea. To check for invalid inputs or wrong results we can use *assert*. In this example, we want to make sure that a given email contains at least one at sign.

```
>>> email1 = "test@testmail.com"
>>> assert "@" in email1, "Invalid input!"
>>> email2 = "email.email.org"
>>> assert "@" in email2, "Invalid input!"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AssertionError: Invalid input!
```

While the first test is fine since @ is included in the given string, this assumption is violated in the second example. Python then stops processing the script at once and throws an

exception so we are informed about the problem. However, note that assert can be used as an internal diagnostic for first checks but make sure to define proper exceptions and especially more testing to sanitise user input. Also, assert statements are removed from the code when performance is optimized by some compilers.

However, sometimes we want to *silence* errors explicitly and continue with the script. This is done using try...except. If an error occurs, we can specify in advance how to handle it. As an example, suppose you want to access a certain index in a list that does not exist. Usually, Python would stop the script and complain.

```
>>> a = [1, 2, 3]
>>> a[20]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

As the list has only three elements, there is no element with index 20 available. However, when we catch this error we can continue with the script.

```
>>> for list in matrix:
>>>     try:
>>>             print(list[20])
>>>     except IndexError:
>>>             print("Index not found, continue")
```

This script takes lists from a given matrix and always displays the element with index 20. Some shorter lists might not contain so many elements, which would cause problems. However, as we can foresee that this error might occur, we define that all IndexErrors will be caught by our script, produce a short warning note and then resume with the code. There is also the possibility to create catch-alls, which are statements that silence any type of error. Be very careful when working with these things and better specify in advance which errors are possible.

Modules

Some functions or objects are always available in Python, for example, lists or the functions *len()* or *max()*. Some other functions are also official parts of Python but are grouped in modules that must be imported before usage. This is an efficient solution since not all functions are always loaded into Python and many more names for variables and functions are available for yourself. To access these other functions we need to import the respective modules. Let’s demonstrate their usage with some mathematical function

```
>>> import math
>>> math.cos(math.pi)
-1.0
```

Here we import the *math* module to access one constant and one function from this module. The prefix *math* is subsequently used to tell Python where to take the functions from. However, typing this all the time can become tedious so there are workarounds. For example, we can shorten the name of a module to make writing and reading code more convenient.

```
>>> import itertools as it
>>> list(it.combinations([1, 2], 2))
[(1, 2), (1, 3), (2, 3)]
```

As long as only a few functions are required you can also only import this function.

```
>>> from itertools import combinations
>>> list(combinations([1, 2], 2))
[(1, 2), (1, 3), (2, 3)]
```

If you need all functions, use the asterisk as a generic placeholder.

```
>>> from itertools import *
>>> list(combinations([1, 2], 2))
[(1, 2), (1, 3), (2, 3)]
```

When working with longer scripts and more complicated tasks it can be especially beneficial to keep the respective module prefixes so it is clear to all colleagues where certain functions are taken from.

1.3 • Principles of Good Programming

1. Indentations play a significant role in Python as they replace most of the parentheses and brackets known from other programming languages. Whether you are using spaces or tabs for indentations is irrelevant as long as you are consistent and never mix them, which causes Python to produce an error message.
2. All variables and objects (and in Python, virtually everything is an object) should have a unique and clear name. There are certain styles to choose from, for example, *Panelleft* (Pascal case), *panelLeft* (Came case), or *panel_left* (Snake case). However,

be consistent with your style. It might not be necessary to waste time thinking about names for index variables (often just i) or very temporary variables. Try to limit the usage of one-letter variables for narrow blocks of code or comprehensions.

3. Functions should usually do exactly *one* thing. If you conclude that a given function does a lot of things, maybe due to the usage of many if...else statements, it might be wise to split it up. Also, never define two functions at two different places in the code that do the same thing but define it once and when call whenever necessary. This makes debugging a lot easier and you have to clean up bugs only in one place if you find any. Furthermore, a function should normally only return one data type (for example, a math function that only returns integers but not strings or lists). Whenever something goes bad, do not return a special "error code" or *False* but raise an exception.[4]
4. Python was created to get things done and work efficiently. Therefore it might be a good idea to think about certain parameters before starting a project. How many people are involved, how much time will it take? Should I start defining ten classes or are a few functions enough to get the job done? Will I work with this code again in five years or is it obsolete next week? Depending on the answers, you might want to spend more time preparing the project and defining things, possibly with your colleagues. This refers to a common style of coding, naming objects, and creating shared documentation. Note that even the smallest projects deserve *some* documentation, even if it is just for a weekend project.
5. Readability is a major factor in any code. For example, consistent spacing makes it much easier to understand. Therefore, I recommend making use of it and writing x = (5 + 5) instead of x=(5+5). Again, there are no strict rules but rather guidelines you can choose. In this book, we will insert a space between most numbers and operators.
6. Clear and meaningful documentation is the gold standard of programming. Especially larger, longer running projects with many co-workers deserve extensive documentation that is understandable to all people working on it after you. And even if you code alone, your future self will be very grateful if you spend just a few minutes on documenting what you did. For example, Python docstrings ("""This is the comment""") are very useful to describe what a function or class is doing. In this book there is little documentation within the coding blocks since everything is explained in detail in the chapters so probably do not use this is a template unless you are willing to explain everything as in a tutorial. For inline comments use the number sign #.
7. When you have little experience with version control software it might be a good idea to spend some time learning about it, especially when you are working on larger or longer running projects. This makes the creation of many documents obsolete that allow you to go back to previous versions of your code (we all know final.py, final2.py, final3.py, …). Basic software that helps you out is *git* or *bazaar*. When collaborating online, try *Github*.
8. Debugging, that is finding and fixing errors and bugs in your code usually takes a large part of your time. An advantage of Python that can never be underestimated is that error messages and exceptions are usually very clear and try to describe what went wrong, which makes finding the problem a lot easier. Sometimes these are trivial errors, like missing parentheses or letters. If an error is unknown to you, just search

4 For more information on clean code, research the works of Robert C. Martin. Youtube provides some excellent presentations.

for it online and things might be a lot clearer. Also, when Python reports a line together with the error, make sure you check the lines before and after if you do not find it in the one reported.

9. There is no rule without exception. The guidelines presented here are just basic principles and not written in stone. There might be good reasons to deviate from them but be sure that these are justified. If you feel too tired or lazy to follow a certain style, it is perhaps time for a break instead of writing sluggish code.
10. If you are looking for more detailed information on style and coding principles, make sure to have a look at the official Python style guide PEP8.[5]

1.4 • Problem-Solving Skills

As stated before, this is not a classical and theoretical textbook but rather is a focus on applications and real-world problem-solving skills. Suppose your boss gives you a task and isn't interested in exactly how you process it as long as you quickly present the results. It's up to you to find out how to do it. All in all, Python is a precious tool for tackling complex tasks. It comes with a wide range of libraries, modules, and packages which in many cases are somewhat related to your specific task and can be easily adapted. Since the performance of modern computers is huge it is nowadays also possible to tackle problems by crunching numbers (Brute-Force solutions) or performing simulations for approximate solutions instead of thinking about an analytical solution that requires a lot of theoretical knowledge, time, and experience. What exactly could such a workflow look like?

First of all, it is relevant to understand the given task or problem and get an overview of the situation. Have you already worked on related challenges in the past? Are there similar problems you know about? Try to deduce the unknown to known things, which is quite easy due to search engines or *Wikipedia*. In many cases, you will find ready to use solutions online that perhaps only require implementation in Python. Sometimes you get lucky and all you need to do is copy a few lines of code. This being said, it is of course not the goal of this book to solve the tasks presented here by searching online and looking for ready to use solutions - this would only train your research skills. Therefore, if you are stuck with a problem and run out of ideas, perhaps just skip to the next task and come back later. The human brain works tirelessly and subconsciously on unsolved problems which can lead to *Heureka*-moments.

After you have a plan in mind it is time to work on the implementation in Python, which is an easy task. As discussed before it is often a good idea to split up complex problems into small chunks that can be easily solved. Using functions as an implementation is then quite convenient. At this stage do not strive for perfection as you probably want a first result quickly, which you can later optimise. Often your boss might be happy with a first approximation as long as it is submitted in time. If you struggle with the implementation phase, it might be beneficial to consult a textbook or guide on the required technique. Since Python comes with so many features it is rarely necessary to reinvent things - be smart and be sure to make use of the available functions and modules - these are tested and approved by the community. The official documentation comes with many examples and serves as

5 Pep8.org

a wonderful guideline and teacher. If you need special functions, it might be wise to invest some time to study the documentation of these external packages, especially when diving deeper into some material and plan to work longer on related projects.

If your first attempt is complete and the code is written, it is time to test. Often code will not work directly as planned, resulting in either a runtime error or an obviously incorrect result. Syntax errors are easily debugged because of the quite specific error messages that Python produces. It might be more challenging to wipe out logical errors in your code that relate more to your algorithms and strategy than the implementation itself. If this happens, first try to individually test each function to reduce the potential source of the problem. Think about cases that are easy to test for correctness and work your way up to more complex inputs. It is justifiable to place temporary print-statements inside the code to observe the state of variables. By adding sleep-statements you can also run the code in slow motion and trace the flow. Although this technique of debugging is often ridiculed, there are good reasons to use it, especially in smallish projects. Of course, a real debugger is way more powerful but often depends on the IDE you use and requires further experience. Python comes with the internal debugging system *pdb*[6] which allows you to follow the execution of your code step by step. To spot logical errors, make sure you explain the principles and algorithms of your code to colleagues. This will force you to spell out clearly what the code is doing, which helps in clarifying your ideas.

If the code runs and is clear of any obvious bugs you can try to optimise it. Especially when you work on longer projects which will run more often, increasing performance and refactoring can be a boon. Then you should try to work on readability, documentation, and performance to make your code better and more enjoyable. This task is often more relaxed since your boss is already happy with the first outcome and there is less pressure. Try identifying overly complex blocks of code and cleanly rewrite them. Add more documentation while you work through it. When working on performance, testing functions individually helps you identify the slow parts which might benefit from different approaches. In this book, we will also talk about measuring runtime speeds and working on optimisation.

6 https://docs.python.org/3.6/library/pdb.html

F
factorial *25*
fibonacci *22*

G
gambling *102, 104*
game of life *92*
generator *26*
genetic code *135*
grid *89*

H
hash function *142*
histogram *125*

I
import *17*
init *97*
integral *74*
islice *38*

J
John Conway *92*

K
knight's tour *70*
kwargs *69*

L
lcs *137*
logistic function *50*
logistic map *51*
loop *11*
lps *135*

M
matrix *90*
module *17*
monotonic *68*
multiprocessing *84*

N
norm *57*
numerical integration *74*

O

object *18*
optimization *103*

P

palindrome *135, 154*
parallelisation *84*
pass *13*
pep8 *20*
pi *30, 80*
pig *104*
population *96*
prime number *26*
process *85*
product *123*
profiling *41*
program flow *140*

Q

queue *84*

R

random *83*
random walk *88*
recursion *22, 24, 36*
recursionlimit *40*
refactoring *21*
resampling *128*
return *15*
roman numerals *147*

S

sample *84*
scalar product *59*
series *31*
simulation *103*
slice *9*

T

time *37, 79*
transpose *161*
trigonometry *31, 88*
try *17*

V

vector *57*

Y

yield *27*
yield from *153*

Z

zip *58*

elektor books

Learn to use Python productively in real-life scenarios at work and in everyday life

Python 3 for Science and Engineering Applications

If you have mastered the basics of Python and are wanting to explore the language in more depth, this book is for you. By means of concrete examples used in different applications, the book illustrates many aspects of programming (e.g. algorithms, recursion, data structures) and helps problem-solving strategies. Including general ideas and solutions, the specifics of Python and how these can be practically applied are discussed.

Python 3 for Science and Engineering Applications includes:

- practical and goal-oriented learning
- basic Python techniques
- modern Python 3.6+ including comprehensions, decorators and generators
- complete code available online
- more than 40 exercises, solutions documented online
- no additional packages or installation required, 100% pure Python

Topics cover:

- identifying large prime numbers and computing Pi
- writing and understanding recursive functions with memorisation
- computing in parallel and utilising all system cores
- processing text data and encrypting messages
- comprehending backtracking and solving Sudokus
- analysing and simulating games of chance to develop optimal winning strategies
- handling genetic code and generating extremely long palindromes

Felix Bittmann is a research associate at the Leibniz Institute for Educational Trajectories and a doctoral candidate at the University of Bamberg, Germany. His research interests include social inequality, the role of education in the course of life, quantitative methods, and the philosophy of science. With a focus on statistical analysis and applied research, Python is an integral and multifunctional tool of his daily workflow.

Elektor International Media BV
www.elektor.com

elektor
design > share > sell